An analysis of the roll-back and blocking operations of three concurrency control mechanisms

by VIJAY KUMAR

University of Missouri at Kansas City
Kansas City, Missouri

ABSTRACT

Transactions in database systems are run concurrently to achieve optimal resource utilization. Concurrent execution of transactions is managed by concurrency control mechanisms for maintaining the database consistency. These mechanisms use activities like transaction roll-backs and transaction blockings for serializing the concurrent execution, and they have significant effect on the performance of database systems; however, their relationship with throughput, workload, and other aspects of the system is unclear. Further it is not clear how the read: write ratio affects the performance. This paper attempts to show the effect of roll-back, blocking, and read: write ratio on the performance of database systems under several different types of workloads. We have used detailed and realistic simulation models to conduct our investigation; and, unlike other performance studies, we have avoided simplifying assumptions as far as possible to include most of the attributes of real database systems. In this study we show that neither a roll-back nor a blocking scheme is consistently better for all types of workloads; they are rather workload sensitive. We also show that it is not the write-only transactions but the read-only transactions that need special treatment for efficient processing. We report that transaction wait-time does not have significant effect on the throughput and the effect of read: write ratio is very short lived. We have introduced a new term Domain of Efficiency (DoE) to explain the behavior of these mechanisms.
INTRODUCTION

Recent performance studies of concurrency control mechanisms (CCMs)\(^1\)\(^2\)\(^3\) have shown that mechanisms based on two-phase strategy outperform others. However, CCMs based on two-phase locking suffer with the problem of deadlock.\(^3\)\(^4\)\(^5\)\(^6\) A deadlock may involve several transactions and must be detected and resolved. A large number of algorithms are available for deadlock detection\(^7\)\(^8\)\(^9\)\(^10\)\(^11\)\(^12\)\(^13\)\(^14\) and for deadlock prevention.\(^15\)\(^16\)\(^17\)\(^18\) Resolution of deadlocks is usually done either by rolling-back or by blocking concurrent transactions. It is not clear which method is suitable for what kind of environment or how these activities (rolling-back and blocking) affect the system behavior. In this paper, we study in detail the effect of transaction blockings and transaction roll-backs on the performance of database systems. Although a great deal of work has been done in this area, we feel that performance models used in these works do not incorporate some important attributes of a real database system. We try to bridge this gap with our detailed study.

We begin our study by constructing detailed simulation models which incorporate most of the attributes of a real database system such as transaction failure and database recovery, deadlock detection and its resolution. We define the common terms used here, describe the CCMs studied in this paper, and provide their comparative study. Next we review previous work done in this area as well as our approach. Next we explain model parameters and the construction of transactions used in our work. We describe simulation models, and the statistical approach taken for data collection and their validation. Finally, we discuss the simulation results and conclude our findings.

TERMS

In this section we define the common terms used in this paper.

**Entity:** We consider a database as a set of entities of fixed size. An entity is a lockable unit.

**Read:** A transaction reads an entity. The contents of the entity is not changed. It is an atomic action.

**Write:** A transaction after reading an entity modifies its contents. It is an atomic action in the sense that the read and modify operations are not separable.

**Conflict:** Operations of any two transactions \(T_1\) and \(T_2\) accessing an entity conflicts if one of the following conditions is satisfied:

1. \(T_1\) is performing a read (write) operation and \(T_2\) wants to perform a write (read) operation. This is read-write (RW) or write-read (WR) conflict.

2. \(T_1\) is performing a write operation and \(T_2\) wants to perform a write operation. This is a write-write (WW) conflict.

**Blocking:** The execution of a transaction is suspended (blocked) temporarily upon encountering a conflict.

**Restart:** A blocked transaction is rescheduled for execution. The point of restart of a blocked transaction depends upon the CCM.

**Roll-Back:** A process in which all the write actions of a transaction are undone. There are two situations under which a transaction is rolled-back:

1. a blocked transaction is rolled-back.
2. an aborted transaction.

**Commit:** A transaction is said to be committed if and only if it has unlocked all its locked entities.

**Degree of Concurrency (DoC):** Total number of active transactions in the system at any instant. For two-phase locking, an active transaction is a transaction with at least one of its lock requests granted, and which is in a state ready to be scheduled for execution. (The term degree of multi-programming has also been used in literatures for DoC.)

**Degree of Cycle:** Total number of transactions involved in a deadlock.

TRANSACTION PROCESSING AND CONCURRENCY CONTROL MECHANISMS

We describe here in detail the mechanisms of the CCMs we have studied. We analyze their behavior to define the domain of our investigation and we use this domain to perform the simulation experiment. The entire processing of transactions can be divided into the following three phases:

**Locking Phase**—The scheduled transactions lock the required entities. Locks can be applied into two different modes: the share mode (read lock) and the exclusive mode (write lock).

**Execution Phase**—The transaction processes the locked entities.

**Release Phase**—The transaction unlocks the locked entities.

Execution of these phases by concurrent transactions depends on the CCM used. Next we explain, in terms of these three phases, how concurrent transactions are processed by the three concurrency control mechanisms we investigated in this work.
Incremental Locking and Simultaneous Release

In this mechanism the first two phases (i.e., locking and execution) are interleaved and the end of the execution phase initiates the release phase. Entities are locked only when demanded by the execution phase. Consequently, only those entities are locked which are actually processed by the transaction, although the transaction might have referenced many more entities in its code.

A conflict is resolved by blocking the transaction which is trying to lock the entity. A blocked transaction retains all the locks it obtained before getting blocked; it is unblocked and resumes its execution when the entity it required becomes free. We also refer to this activity as transaction restart. Resolving conflicts by blocking a transaction may create deadlock which must be resolved by rolling-back a transaction. A rolled-back transaction may be rerun or may be removed from the system. At the end of the execution phase all the entities locked by the transaction are released in one atomic action (simultaneously). In this report we refer to this mechanism as D1 protocol.

Wait-Die (WD) and Wound-Wait (WW) Mechanisms

Rosencrantz and others\textsuperscript{9} have discussed Wait-Die and Wound-Wait mechanisms which can be described as follows:

Let \( T_1 \) and \( T_2 \) be two transactions and \((T_1, -ts)\) and \((T_2, -ts)\) their associated timestamps. Suppose that \( T_1 \) makes a lock request (requester) for an entity \( E \) currently locked by \( T_2 \) (holder), thus generating a conflict over \( E \).

Such conflict is resolved by these techniques as follows:

a. The Wait-Die System (WE Protocol)

If \((T_1, -ts) < (T_2, -ts)\) then wait else die. If the requester's timestamp is smaller, then it waits for the holder—otherwise, the requester dies (roll-backs).

b. The Wound-Wait System (WW Protocol)

If \((T_1, -ts) < (T_2, -ts)\) then wound else wait. If the requester's timestamp is smaller, then wound the holder—otherwise, wait for the holder \( (T_1 \) wounds \( T_2 ) \) to release the entity.

Comparison of Protocols

In this section we hypothesize the expected behavior of the three mechanisms and later we verify the hypotheses. As mentioned earlier, the strict two-phase (D1) locking resolves conflicts by blocking transactions. Blocking transactions does two things: increases transaction wait-time (amount of time transaction remains blocked) and creates the possibility of deadlocks. A blocked transaction reduces the availability of entities for other transactions; the higher the number of blockings the lower the availability of entities. Reduction in the availability of entities would in turn increase transaction blockings which might increase the probability of deadlock occurrence. This increase, however, remains very low and becomes significant only at very high transaction arrival rates.\textsuperscript{20} An increase in deadlock increases transaction roll-backs. Thus we see that these activities are interdependent and a change in one affects in some way all the other activities.

It seems that if transaction wait-time is reduced, some improvements in system performance may be achieved. One of the ways this can be done is by rolling-back transactions instead of blocking them. This approach is taken in WD and WW, which assume that blocking a transaction is likely to create a deadlock and hence they roll-back the transaction when a conflict arises. On one hand this policy increases the availability of entities and minimizes transaction wait-time, but on the other hand it consumes, comparatively, more CPU and I/O resources and also rolls-back transactions which may never cause a deadlock. We refer to this type of roll-back as redundant roll-backs.

It has been shown\textsuperscript{20} that even at very high arrival rates the degree of cycle (number of transactions in a cycle) remains low. In this situation it would seem that redundant roll-backs are expensive and might get worse for larger transactions. Another important factor which should be considered is the size of roll-back (number of entities to be restored in a roll-back). If the average roll-back size is small, then the process would not be expensive. Blocking a transaction which is holding one entity for some length of time is certainly more expensive than rolling it back. The latter is even less expensive if the entity happens to be in the main memory.

Intuitively it seems that if the average transaction size is large then the average roll-back size may be large and may become expensive. We aim to verify if there is any relationship between the average transaction size and the size of roll-back. We aim to compute the Domain of Efficiency (DoE) of these mechanisms. We define the DoE of a CCM as the set of ranges of those parameter values inside which there exists a linear relationship among these parameters. For example, we might discover that in a CCM up to an average transaction size of \( N \), the average roll-back size increases linearly. When the average transaction size becomes larger than \( N \), this relationship may become unpredictable. The relationship among other parameters may be observed in a similar way to define the DoE of a CCM. We aim to establish the set of parameters and their acceptable ranges.

In D1 a roll-back is not instantaneous. It can take place only when the deadlock is detected and resolved. There are several criteria for selecting a transaction to be rolled-back to resolve a deadlock;\textsuperscript{21} we have selected the youngest transaction for this purpose. A blocked transaction resumes its execution when other transactions are rolled-back or committed. Since these operations are not instantaneous in D1, transaction wait-time will keep on increasing. In contrast, in WD and WW a roll-back may be instantaneous, and blocked transactions do not wait longer than they do in D1. In WW the number of blocking is much higher and these blocked (waiting) transactions may get wounded. In WD mechanism, a blocked transaction never dies: either a transaction's waiting-time is increased or, in the case of its death, its recovery time is increased but never both.

We aim to verify these points in this work.
REVIEW OF PREVIOUS WORK

The database literature is full of performance reports. Each report emphasizes certain aspects of some mechanisms. In Kiessling and Landherr22 the effect of roll-backs and transaction blockings have been looked into in a limited way. They do not take into account the effect of deadlocks on throughput and they assume that the entire database is divided into 100 granules, which is not very realistic. Also, a performance comparison of timestamping and strict two-phase locking has been done.23 They report that when average transaction size is small (4–8 entities), resolving conflicts by rolling-back is better than blocking transactions and for larger transactions rolling-back becomes too expensive. The report contains few details about the effect of varying transaction arrival rates and different types of workloads. Tay, Suri, and Goodman24,25 report that in all the situations, the no-waiting case performs better than the waiting case. Their studies show that transaction blockings have a significant detrimental effect on the system throughput and concludes that locking with no waiting seems to be a practical approach, if the cost of transaction restarts is brought down to a minimum. They do not say how this can be achieved. Ries and Stonebraker26 reported that the optimistic method (conflicts are resolved by rolling-back transactions) always performed better. A possible reason for this result could be that they assumed the roll-back cost to be negligible. Further, a detailed study of deadlock has been done27 in which the authors have basically looked at the effect of transaction blocking and restarts on system performance and tried to find a deadlock treatment technique which is consistently better. They report that there is no such deadlock treatment strategy that performs consistently better and the performance of a strategy very much depends on the type of workload. They found that in a situation in which resources are fairly heavily utilized, continuous deadlock detection is preferable, but in Kumar28 it is reported that even at high transaction arrival rates deadlock occurrence is very low and a continuous deadlock detection does have some effect on the performance.

Our Approach

Our model has its origin in the model of Ries,24 however, we have avoided simplifying assumptions as far as possible to simulate a real system as precisely as possible. In almost all the work we have reviewed, no one has included transaction failure and they have assumed that the IO and CPU requirements of transactions depend on their size. Also, in some reports they have simulated deadlocks, transaction blockings, and restarts simply by introducing estimated delays. Most of these works have used only one type of workload to drive their simulators except one29 in which three different types of workloads have been simulated.

It can be argued convincingly that these simplifying assumptions do affect the simulation, and the results so obtained may fail to explain the behavior of CCMs correctly. In our investigation we have included most of the attributes of a real system to study the behavior of these CCMs precisely. For this reason we feel that a direct comparison of our work with others would not be very meaningful.

We have studied the performance of D1, WD and WW under strictly identical environments. We selected D1 protocol to avoid cascade roll-backs and the same locking policy (incremental locking) as D1 was used in WD and WW mechanisms for a meaningful comparison of transaction roll-back and blocking. We have coded the deadlock detection and resolution, database recovery, transaction blocking, and transaction roll-back algorithms as they would exist in a real database system. In this respect our system, to a large extent, represents a real database system. On this basis we claim that our results may be more reliable and informative than the results obtained in other reports.

The assumptions we have made to build our models follow:

1. There is one IO processor and one CPU.
2. Transaction IO requirements are partially and transaction CPU requirements are totally independent of their size. This means that the probability that a large transaction would use more IO than a small transaction is high, and two same size transactions may not use the same amount of IO.
3. The CPU resource is shared between locking and processing activities.
4. Recovery operation is required in the case of transaction failure and in deadlock resolution, and has been given the highest priority (i.e., a transaction to be rolled-back goes to the top of processing queues and is processed in the next event) #34.
5. Our models are open-ended, i.e., a set of transactions does not cycle in the system for repeated execution. Also a rolled-back transaction is not rescheduled. We used open-ended models to study the worst case behavior of the CCMs we tested.
6. Transaction failure is unpredictable. A transaction can be aborted by the user, it can fail due to disk fault or due to some other system problems. In [32] it is reported that transactions failures are not frequent and the failure percentage lies usually between 2–3%, so we assumed that 2% of transactions would fail in a simulation run.
7. Lock table always resides in the main memory.

MODEL PARAMETERS

The system parameters for our models are:

1. IO processing time (time taken by IO processor to transfer an entity from the disk)
2. CPU processing time (time taken by the CPU to process an entity)
3. CPU locking time (time taken by the CPU to lock an entity)
4. CPU recovery time (time taken by the CPU to restore the last consistent state of an entity)
5. Node check time (time taken by the CPU to check one node in a wait-for graph)
6. Deadlock detection cost (time taken by the CPU to detect a deadlock)
7. Cycle detection frequency (defines after how many transaction blockings the deadlock detection is performed)

Input Parameter

To generate the three different transaction size mixes we have used the approach taken in Ries and Stonebraker:28

1. All transactions are roughly the same size. This transaction mix is generated by uniform distribution.
2. About 35–40 percent of transactions are large and the rest are small (accessing about 25–30 percent of average transaction size). This transaction mix is generated by exponential distribution.
3. About 95–97 percent of transactions are very small (accessing maximum number of entities). This transaction mix is generated by hyper-exponential distribution.

We have assumed that most of the real transaction processing environment would fall into the defined domain. We do not claim that these transaction mixes completely represent the real transaction processing environment; however, they have some flavor of such an environment.

Workload Parameters

1. Total number of transactions to be processed.
2. Database size.
3. Read : Write ratio. We define the read : write ratio of a transaction as the ratio of the number of read actions and the number of write actions. Under this scenario, if a transaction size is 30 and its read : write ratio is 20:10, then the transaction will perform 20 reads and 10 writes. The selection of entities for read and write operations are done randomly under a uniform distribution. Thus any entity out of 30 entities is equally likely to be selected for a read or write operation. To study the effect of this ratio, we start our simulator with read-only transactions (read : write ratio is $m : 0$; where $m$ is transaction size and it also indicates the number of read operations in a transaction). We collect all the statistics for this run. The same set of transactions are then run, each with $n$ number of write locks, such that the read : write ratio is $m - n : n$, where $n = 1, 2, 3, \ldots, m$, and statistics are collected for all the runs. When $m = n$ then all transactions of the set become write-only transactions.
4. Percentage of transaction failure. We have modeled this kind of failure by parameterizing the failure percentage. A required failure percentage can be supplied and the system randomly selects so many transactions (victims) which will fail during execution. The failure points of these victims are selected randomly and represented in terms of sub-transactions which would be processed successfully before the transaction fails. For example, if the total number of sub-transactions in a transaction is 10, then the failure point may be any number between 1 and 10. In this case the transaction will fail after processing so many sub-transactions and will then be scheduled for roll-back (recovery).

Output Parameters

1. Average number of transaction restarts. Transaction restart means the resumption of the execution of blocked transactions. A blocking point may be anywhere in the transaction (i.e., a transaction may get blocked at its first lock request or it may get blocked after occupying some locks).
2. Degree of cycle.
3. Throughput.
4. Average response time.
5. Average Recovery Time.
6. Degree of Concurrency (DoC).
7. Average number of deadlocks.
8. Average transaction roll-back size.
10. Average wait-time of wounded/dead transactions.

CONSTRUCTION AND SCHEDULING OF TRANSACTIONS

A transaction for our simulation is viewed as an ordered set of several sub-transactions, each of which holds a certain number of entities as shown below:

$$T(m) = \{t_1(e_1), t_2(e_2), \ldots, t_m(e_m)\}$$

where $T =$ parent transaction, $m =$ total number of entities required by $T$, $t_1, t_2, \ldots, t_m =$ sub-transactions, $e_i =$ number of entities required by $t_i$, and

$$m = \sum_i e_i$$

During the locking and execution phases of $T$, $t_1$ locks $e_1$ entities and processes them, then $t_2$ locks $e_2$ entities and processes them, and so on. When $t_m$ has successfully locked and processed $e_m$ entities, $m$ entities are released in one atomic action.

SIMULATION MODELS AND EXECUTION OF TRANSACTIONS

The simulation models are shown in Figures 1 and 2. The dotted lines represent the use of CPU and IO resources by the related activities. The flow of transaction (execution) through it is as follows:
1. A transaction arrives (Poisson arrival) and joins the pending queue (PQ).
2. A transaction is picked up from the top of PQ, the total number of entities required by this transaction is calculated, and its sub-transactions are created by a random process. Another process randomly distributes the total entities of this transaction among its sub-transactions. A FIFO is used in processing the PQ.
3. The first/next sub-transaction of the parent transaction requests the required locks one at a time.
4. If any of the locks of that sub-transaction is denied, then, depending upon the CCM being simulated, a proper action (roll-back or blocking) is taken to resolve the conflict. A blocked transaction goes to blocked transaction queue (BTQ). In the case of D1, a deadlock detection is initiated after every conflict or after a pre-specified number (can be specified via a parameter) of transactions are blocked. If a deadlock is found, then the cycle is broken by rolling-back a transaction from the cycle. Several transaction selection criteria can be specified by a parameter; we selected the youngest transaction for this purpose. As mentioned earlier, a roll-back process is given the highest priority, so such a transaction is put at the top of IOQ and is processed before all those transactions which joined IOQ before it.
5. After a successful locking phase, a transaction goes through a random victim selection process. The parameters of this selection process are set so that it achieves a 2% transaction failure. A transaction goes through this process only once.
6. IO request of a sub-transaction is processed.
7. CPU request of a sub-transaction is processed.
8. If more sub-transactions of a parent transaction are left to be processed, then the parent transaction is moved to the bottom of PQ.
9. At the end of processing all the sub-transactions, the parent transaction is committed and all BTQ transaction are moved to PQ and resume their normal processing.

Simulation Parameters Values

All three simulators have been tested with the following parameter settings. One simulation time unit may be interpreted as one millisecond.

Transaction arrival rate range: 0.5–20 transactions/second.
Read:write ratio range: \( m : n \) : \( n \) (\( n = 0, 1, 2, \ldots, m \)).
Victim selection parameter: set to give 2% transaction failure.
Database size: 3000 entities.
(CPU processing time: 2.50 time unit/entity.
CPU locking time: 1.05 time unit/entity.
CPU recovery time: 3.55 time unit/entity.
Deadlock detection cost: 1.05 time unit.
IO processing time: 25.15 time unit/entity.
Cycle detection frequency: set to perform after 1, 10 and 15 conflicts.

The parameter values were chosen so as to be able to simulate from lightly loaded system (one or less active transaction in the system) to heavily loaded system (more than 25 active transactions in the system). Using the rule of thumb proposed in Tay, Goodman and Suri we expect our system to start thrashing at heavy workloads, which in our case is likely to occur when DoC is between 25 and 30.

The Statistical Approach

We describe in this section the statistical approach we took to discriminate between throughput differences owing simply to statistical variations and those actually owing to algorithm performance characteristics.
Several simulation analysis techniques are available (a survey of these techniques may be found in 31, 32, 33). We selected the method of batch means from the options of batch means, regenerative method and the independent replications. In our test runs we found that following initial set-up, an idle state, where all transactions input sources (terminals) are in their stagger delay does not occur with sufficient frequency to allow the use of regenerative method. The batch means has the advantage over independent replication that initial transients do not bias each of the throughput observations. From the implementation viewpoint, the batch means method is simpler than the method of replication since in the latter the simulator has a garbage collect and reinitialize simulation and the data structures between the observation periods.

Under the selected method (batch means) a simulation run is divided into a set of batches. Each batch is a fixed simulation time-units long and each such batch provides one throughput observation. All individual throughput observations are averaged to estimate the overall throughput. Standard techniques with the assumption that the throughput observations from the batches are independent and identically distributed are used to compute the confidence intervals. In our simulation run we used the following values of batch-num and batch-time:

\[
\text{Batch\_num} = 20; \quad \text{Batch\_time} = 50000; \quad \text{Total simulation time units} = 20 \times 50000
\]

RESULTS AND DISCUSSION

We present the results of only two types of transaction size distributions: uniform and hyper-exponential. The results of exponential distribution follow a similar pattern as the uniform distribution except they have higher values of output parameters. We have taken the liberty of mentioning the results of exponential distribution without providing graphs for them.

Arrival Rate Versus Deaths, Restarts and Wounds

The graphs in Figures 3 and 4 show the relationship between average transaction arrival rate and number of deaths (WD), number of transactions wounded (WW) and the number of transaction restarts (D1). Figure 5 shows the relationship between the arrival rate and transaction restarts for all the three mechanisms.

The number of restarts in D1 increases comparatively rapidly with arrival rate. In the hyper-exponential distribution (Figure 4) the increase is even sharper. At lower arrival rates the number of restarts in all the CCMs is nearly the same and it widens after an arrival rate of 12 transactions per second on the average. A comparison of D1 and WW mechanisms indicates that in WW only the younger transactions are blocked to resolve conflict, since the younger requesters never cause deadlocks to occur. At higher arrival rates D1 suffers with deadlock occurrences which increase the transaction blockings and WW with wounded transactions which reduce the number of such blockings. The situation in WD is just the opposite; all younger transactions are rolled-back to resolve conflicts and consequently the number of restarts remains quite low even at higher arrival rates. A low number of restarts is bound to increase the number of roll-backs (deaths) and that is shown to happen (Figures 3, 4, and 5). The number of restarts is the highest in D1 and the lowest in WD, and the number of roll-backs is the highest in WD.
Transaction Wait-Time and Restarts

To evaluate the effect of restarts we look at the average time a transaction has to wait (transaction wait-time) for the required entity to become available. As stated earlier, one of the aims of CCMs is to minimize the wait-time. Figure 6 shows the relationship between average transaction wait-time and arrival rate. At very low arrival rates there is no significant difference in this parameter value for these CCMs. The difference between D1 and the other two mechanisms widens after the arrival rate of 11 transactions per second on the average. One noticeable result in Figure 6 is the wait-time for read-only transactions under these CCMs. Under WD and WW, transactions with 0 or 2 write locks wait longer than transactions with higher values of write locks.

Read-only transactions experience wait only in IO and CPU queues. Every transaction after lock requests joins IO queue and waits there for its turn. How long a transaction would wait in IO queue depends totally on the efficiency of IO processor. In D1, as the number of write locks increases so does the number restarts, and a transaction may experience wait at three places: in blocked queue, in IO queue, and in CPU queue. The situation is different in WD and WW mechanisms. As the number of write locks increases, so does the number of conflicts. The increase in the number of conflicts increases transaction deaths or the number of wounded transactions where some transactions would never reach IO queue, and these do not block the IO activity of other transactions. To verify this unexpected result we measured the average waiting-time experienced only by successfully completed transactions and found a further reduction in the waiting-time. We also looked at the waiting-time experienced by wounded and dead transactions (see Figure 7). As expected, on the average a wounded transaction experiences much higher waiting-time than a dead transaction, simply because a blocked transaction in WW is likely to get wounded. This implies that for lower values of the read:write ratio (higher number of write locks in a transaction) WD and WW mechanisms minimize transaction waiting-time, but transaction waiting-time increases as the value of this ratio increases (higher number of read locks).

Figure 7 shows the relationship between the arrival rate and average transaction wait-time of only wounded/dead transactions. The wait-time is higher in WW than it is in WD, which is expected since in WW a higher number of transactions are blocked than in WD (see Figure 4). A higher number of restarts in WD keeps the transaction wait-time to a minimum.

Since transaction restarts increase wait-time, and transaction roll-backs reduce this time, we next look at the average transaction roll-back size. Intuitively, if this size is large then transaction roll-backs would gradually become expensive as the average transaction size increases.
Arrival Rate and Roll-Back Size

As mentioned earlier, average roll-back size is the number of entities required to be restored to get the last consistent state of the database. Figures 8 and 9 show the relationship between the arrival rate and average roll-back size. In WD and WW the average roll-back size decreases with arrival rate. At higher arrival rates, on the average, fewer number of entities are restored because at lower arrival rates a transaction manages to lock more number of entities before it conflicts with other transactions. The probability of conflict increases with the arrival rate, consequently, transactions begin to conflict with other transactions sooner and the average roll-back size begins to shrink.

In the hyper-exponential distribution (Figure 9), the region of higher roll-back size for the WW mechanism begins from the arrival rate of 10 transactions per second on the average. This size begins to decline slowly after the arrival rate of 18 transactions per second on the average. The situation is slightly different in the WD mechanism, where the average roll-back size shows a slight increase after the arrival rate of 17 transactions per second. It seems that under WD between the arrival rate of 10 through 17 transactions, almost all very large transactions go through successfully, keeping the roll-back size to a minimum. After this arrival rate, however, large transactions get rolled-back and the roll-back size increases.

In WW, it seems that most of these large transactions face roll-back between the arrival rate of 10 through 18 transactions per second on the average. This is possible since in WW a blocked transaction can also be rolled-back at any time.

Deadlocks and Degree of Cycle

Deadlocks do not reduce the DoE drastically. Table I lists the number of deadlocks at various transaction arrival rates for different read:write ratios. We observe that the number of deadlocks is affected more by the type of workload than by the arrival rate. In hyper-exponential distribution, the frequency of deadlock occurrence is higher, and they also occur at lower arrival rates.

The problem of deadlock would have stronger effect, if the degree of cycle is large. We observe in Figure 10 that the degree of cycle at all arrival rates remains low. In fact in our investigation, in uniform distribution, it never exceeded three transactions. At an arrival rate of 20 transactions per second with write-only transactions we observed 8 deadlocks when about 500 to 600 transactions are run. The average degree of cycle never exceeded 2 transactions. This is low when we consider the transaction processing environment. However, its effect on throughput is noticeable. The picture is somewhat

<table>
<thead>
<tr>
<th>Average Arrival Rate</th>
<th>Transaction Size Distribution Uniform (No. of Write Locks)</th>
<th>Transaction Size Distribution Hyper-Exponential (No. of Write Locks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2W 6W 10W 14W 18W 22W 26W 30W 2W 6W 10W 500W</td>
<td>2 2 3 2 3 4 2 3 4</td>
<td>1 2 3 4 1 2 3 5 1 2 3 4 1 2 3 4</td>
</tr>
<tr>
<td>.006 .007 .008 .009 .010 .011 .012 .013</td>
<td>1 2 3 4</td>
<td>1 2 3 4 1 2 3 5 1 2 3 4 1 2 3 4</td>
</tr>
<tr>
<td>.014 .015 .016 .017 .018 .019</td>
<td>1 2 3 4 1 2 3 4</td>
<td>1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4</td>
</tr>
<tr>
<td>.020 .021 .022</td>
<td>1 2 3 4 1 2 3 4</td>
<td>1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4</td>
</tr>
</tbody>
</table>
different in hyper-exponential distribution. The degree of cycle here shows sudden increase between the arrival rates of 10 to 16 transactions per second. We believe these peaks are caused mainly by the very large transactions. It seems that within this range of arrival rates, large transactions acquire a large number of entities, raising the probability of conflict among transactions. At higher rates outside this range, a higher number of very small transactions manage to acquire their locks without conflicting with large transactions.

**Read : Write Ratio**

One common observation is that the behavior of read-only transactions is more noticeable than transactions with some write locks. When the number of write locks varies from 10 onward, the behavior of transactions does not change significantly. Our results show that write-only transactions and read-only transactions show distinct behavior, and transactions with all intermediate values of this ratio exhibit nearly the same behavior. We do not think the read : write ratio plays a significant role in shaping the performance of a CCM.

**Arrival Rate, System Throughput, and Boundary of Efficiency**

Figures 11 and 12 show the relationship between the system throughput and the arrival rate for uniform and hyper-exponential distributions respectively. At higher arrival rates, the throughput of D1 compared to WD and WW declines rapidly. As noted earlier, D1’s performance suffers with high transaction wait-time due to transaction blockings. At lower arrival rates, the probability of deadlock occurrence is nil (see Table I) and the number of restarts is high (see Figure 5). As the arrival rate increases, the throughput begins to decline because of the higher restarts and deadlocks. Eventually, after the arrival rate of 20 transactions per second on the average (Figure 11), its throughput becomes the lowest. In the case of hyper-exponential distribution, the decline begins to take place after 18 transactions per second on the average. A glance at Figure 12 shows that at that rate the average number of deadlocks and the degree of cycle are higher than they are in the uniform distribution (Figure 11). To see the effect of larger size transactions, we repeated the same experiment with exponential size distribution of transactions. We found that as the transaction size increases the throughput declines more rapidly due to the increased number of restarts and deadlocks. However, at lower transaction arrival rates (lower than 20 transactions per second on the average (Figure 11), and 18 transactions per second (Figure 12), D1 performs better than WW and WD. It seems then that the domain of efficiency (DoE) of D1 is narrow compared to WW and WD, and D1 should be used for smaller size transactions under certain arrival rates.
The throughput of WD is between WW and D1. The main factor affecting the throughput of WD is the higher number of deaths (Figures 3 and 4). Although the average roll-back size remains low (Figures 8 and 9) its effect predominates from 15 to 16 transactions per second onward. It is interesting to note that a large reduction in transaction wait-time (Figure 6) does not seem as effective as we expected. It should also be noted that the effect of deaths begins to show as early as an arrival rate of 10 transactions per second on the average. The throughput of WW is the highest in the two transaction size distributions. The strategy of WW can be said to be a compromise between excessive transaction roll-backs and transaction wait-time. Similar results have been reported. The average number of wounds is smaller than the number of deaths in WD, but the transaction wait-time is higher due to higher transaction blocking. A combination of these two seems to be more effective in enlarging the domain of efficiency of WW.

To test the consistency of DoEs of D1, WD, and WW, we tested these mechanisms under extreme cases, that is, large transaction sizes (100 entities on the average) and very high transaction arrival rates (100 transactions per second). In these cases we discovered that WD is comparatively more sensitive to these variations than the other two. In particular, the rate of decline of throughput was much sharper in WW compared to D1 and WD. The reason seems to be the slow increase in the average roll-back size and the frequency of deadlock occurrence.

The last point we would like to mention is that the degree of concurrency (level of multiprogramming) does not have any direct relationship with the DoE. The larger the degree of concurrency does not necessarily mean wider DoE, because after a certain value of degree of concurrency the system starts thrashing and the throughput begins to decline rapidly.

CONCLUSIONS

We have investigated the effect of transaction roll-backs and transaction restarts on the performance of the system. Our aim was to define in some clear terms the DoE of two-phase concurrency control mechanisms under varied transaction processing environments. We observed that the DoE is sensitive to a number of parameters. The degree of sensitivity varies among CCMs, which makes us unable to claim one CCM to be uniformly superior to others. In the case of D1, we found that the effect of deadlock is negligible up to a certain transaction arrival rate and after this rate the efficiency of D1 declines, making it inferior to WD and WW. The largest DoE is offered by WW; however, in extreme cases it appears to be more sensitive to transaction size than D1 and WD. The DoE declines more rapidly here when the average transaction size is increased. We conclude that under the worst transaction processing environment performance of WW is the worst.

The DoE of WD can be placed between D1 and WW and we found it to be the least sensitive. It does not vary significantly when other parameters such as transaction size and arrival rates are varied. However, the performance of WD is superior to D1 only at higher arrival rates.

Read-only transactions have the same performance results under all the CCMs. We notice that read-only transactions experience longest transaction wait-time in WW and WD. The situation is just the opposite in D1, where read-only transactions experience the least amount of wait.

We contradict our earlier observation that a reduction in transaction wait-time would considerably improve the performance of CCMs. A balance between the wait-time and the number of restarts seems to be more suitable for improving performance.

We notice that deadlocks are not so harmful and their detection and resolution are not so damaging to the resource utilization. In our opinion D1 should be preferred for a lightly loaded system and WD for a heavily loaded system.

REFERENCES


