# Concurrency Control in Database Systems

Bharat Bhargava, *Fellow*, *IEEE*

**Abstract**—Ideas that are used in the design, development, and performance of concurrency control mechanisms have been summarized. The locking, time-stamp, optimistic-based mechanisms are included. The ideas of validation in optimistic approach are presented in some detail. The degree of concurrency and classes of serializability for various algorithms have been presented. Questions that relate arrival rate of transactions with degree of concurrency and performance have been briefly presented. Finally, several useful ideas for increasing concurrency have been summarized. They include flexible transactions, adaptability, prewrites, multidimensional timestamps, and relaxation of two-phase locking.

**Index Terms**—Degree of concurrency, adaptability, time-stamp, optimistic, classes of serializability, performance, flexible transactions.

## 1 Introduction

Database systems are essential for many applications, ranging from space station operations to automatic teller machines. A database state represents the values of the database objects that represent some real-world entity. The database state is changed by the execution of a user transaction. Individual transactions running in isolation are assumed to be correct. When multiple users access multiple database objects residing on multiple sites in a distributed database system, the problem of concurrency control arises.

The database system through a scheduler must monitor, examine, and control the concurrent accesses so that the overall correctness of the database is maintained. There are two criteria for defining the correctness of a database: database integrity and serializability [5]. The database integrity is satisfied by assigning a set of constraints (predicates or rules) that must be satisfied for a database to be correct. The serializability ensures that database transitions from one state to the other are based on a serial execution of all transactions. For formal definitions, we refer the reader to Appendix A.

Concurrency control in database system has been the focus of research in the past 20 years. Concurrency control problems and solutions have been formalized in [24] and implemented and used in a variety of real world applications [17].

In this paper, we present several classes of concurrency control approaches and present a short survey of ideas that have been used for designing concurrency control algorithms. We have presented ideas that give an insight in the performance of these algorithms. Finally we present a few ideas that are useful in increasing the degree of concurrency.

## 2 Concurrency Control Approaches and Algorithms

Our main concern in designing a concurrency control algorithm is to correctly process transactions that are in conflict. Each transaction has a read set and a write set. Two transactions *conflict* if the read set of one transaction intersects with the write set of the other transaction and/or the write set of one transaction conflicts with the write set of the other transaction. We illustrate this further in Fig. 1.

If read set $S(R_1)$ and write set $S(W_2)$ have some database entities (or items) in common, we say that the read set of $T_1$ conflicts with the write set of $T_2$. This is represented by the diagonal edge in the figure. Similarly if $S(R_2)$ and $S(W_1)$ have some database items in common, we draw the other diagonal edge.

If $S(W_1)$ and $S(W_2)$ have some database items in common, we say that the write set of $T_1$ conflicts with the write set of $T_2$. This situation is represented by the horizontal edge at the bottom.

We do not need to worry about the conflict between the read sets of the two transactions because read actions do not change the values of the database entities.

It must be noted that transactions $T_1$ and $T_2$ can conflict only if both are executing at the same time. If, for example, $T_1$ has finished before $T_2$ was submitted to the system, even if their read and write sets intersect, they are not considered to be in conflict.

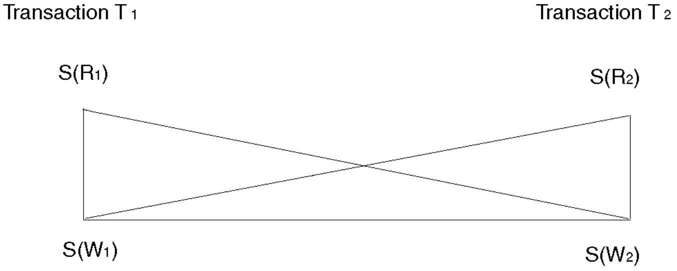


Fig. 1. Types of conflicts for two transactions.

- *B. Bhargava is with the Department of Computer Sciences, Purdue University, West Lafayette, IN 47907. E-mail: bb@cs.purdue.edu.*

*Manuscript received 7 July 1997; revised 26 Jan. 1999.*
*For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number 109082.*

### *Generic Approaches to Synchronization*

There are basically three generic approaches that can be used to design concurrency control algorithms. The synchronization can be accomplished by utilizing:

- **Wait**: If two transactions conflict, conflicting actions of one transaction must wait until the actions of the other transactions are completed.
- **Timestamp**: The order in which transactions are executed is selected based on a time stamp. Each transaction is assigned a unique timestamp by the system and conflicting actions of two transactions are processed in timestamp order. The time stamp may be assigned in the beginning, middle or end of the execution. Version-based approaches assign time stamps to database objects.
- **Rollback**: If two transactions conflict, some actions of a transaction are undone or rolled back or else one of the transactions is restarted. This approach is also called *optimistic* because it is expected that conflicts are such that only a few transactions would rollback.

In the following section, we give further details of each of these approaches and describe the concurrency control algorithms that are based on them.

## 2.1 Algorithms Based on Wait Mechanism

When two transactions conflict, one solution is to make one transaction wait until the other transaction has released the entities common to both. To implement this, the system can provide *locks* on the database entities. Transactions can get a lock on an entity from the system, keep it as long as the particular entity is begin operated upon, and then give the lock back. If a transaction requests the system for a lock on an entity, and the lock has been given to some other transaction, the requesting transaction must wait. To reduce the waiting time when a transaction wants to read, there are two types of locks that can be employed, based on whether the transaction wants to do a read operation or a write operation on an entity:

1) *Readlock*: The transaction locks the entity in a shared mode. Any other transaction waiting to read the same entity can also obtain a readlock.
2) *Writelock*: The transaction locks the entity in an exclusive mode. If one transaction wants to write on an entity, no other transaction may get either a readlock or a writelock.

When we say lock, it means any of the above types of lock. After a transaction has finished operations on an entity, the transaction can do an *unlock* operation. After an unlock operation, either type of lock is released, and the entity is made available to other transactions that may be waiting.

It is important to note that lock and unlock operations can be embedded in a transaction by the user or be transparent to the transaction. In the later case, the system takes the responsibility of correctly granting and enforcing lock and unlock operations for each transaction.

Locking an entity gives rise to two new problems: *livelock* and *deadlock*. Livelock occurs when a transaction repeatedly fails to obtain a lock. Deadlock occurs when various transactions attempt locks on several entities simultaneously; each transaction gets a lock on a different entity and waits for the other transactions to release the lock on the entities that they have succeeded in securing.

The problem of deadlock can be resolved by the following approaches, among others:

- Each transaction locks all entities at once. If some locks are held by some other transaction, then the transaction releases any locks that it was able to obtain.
- Assign an arbitrary linear ordering to the items, and require all transactions to request locks in this order.

Gray and Reuter [17] has described experiments in which it was observed that deadlocks in database systems are very rare and it may be cheaper to detect and resolve them rather than to avoid them.

Since the correctness criterion for concurrently processing several transactions is serializability, locking must be done correctly to assure the above property. One simple protocol that all transactions can obey to ensure serializability is called *Two-phase Locking* (2PL). The protocol simply requires that in any transaction, all locks must precede all unlocks. A transaction operates in two phases: The first phase is the locking phase, and the second phase is the unlocking phase. The first phase can also be considered as the growing phase, in which a transaction obtains more and more locks without releasing any. By releasing a lock, the transaction is considered to have entered the shrinking phase. During the shrinking phase the transaction releases more and more locks and is prohibited from obtaining additional locks. When the transaction terminates, all remaining locks are automatically released. The instance just before the release of the first lock is called *lockpoint*. The two phases and lockpoint are illustrated in Fig. 2.

We now present a simple centralized algorithm that utilizes locking in a distributed database system. For the sake of simplicity, we may assume that all transactions write into the database and the database is fully replicated. In real systems it might be very inefficient to have a fully replicated database. Moreover, the majority of the transactions usually only read from the database. But since multiple copies of a given entity and the write operations of transactions are the major reason for studying concurrency control algorithms, we focus on these issues.

### *2.1.1 A Sample Centralized Locking Concurrency Control Algorithm*

A brief outline of a simple centralized locking algorithm is given below. When a transaction $T_i$ arrives at node $X$, the following steps are performed:

1) Node $X$ requests from the central node the locks for all the entities referenced by the transaction.
2) The central node checks all the requested locks. If some entity is already locked by another transaction, then the request is queued. There is a queue for each entity and the request waits in one queue at a time.

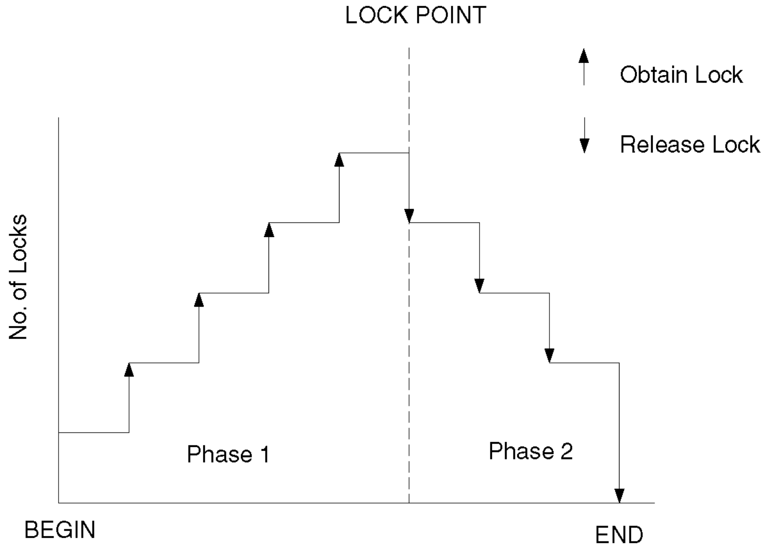


Fig. 2. Two-phase locking and lockpoint: ↑ Obtain lock; ↓ Release lock.

3) When the transaction gets all its locks, it is executed at the central node (the execution can also take place at node $X$, but that may require more messages). The values of read set are read from the database, necessary computations are carried out, and the values of the write set are written in the database at the central node.
4) The values of the write set are transmitted by the central node to all other nodes (if the database is fully replicated).
5) Each node receives the new write set and updates the database; then an acknowledgment is sent back to the central node.
6) When the central node receives acknowledgments from all other nodes in the system, it knows that transaction $T_i$ has been completed at all nodes. The central node releases the locks and starts processing the next transaction.

Some interesting variations of the centralized locking algorithm are as follows:

- *Locking at Central Node, Execution at all Nodes.* Instead of executing the transaction at the central node, we can only assign the locks at the central node and send the transaction back to node $X$. The transaction $T_i$ is executed at node $X$. The values of the read set are read, and the values of the write set are obtained at node $X$. Node $X$ sends the values of the write set and obtains acknowledgments from all other nodes. It then knows that transaction $T_i$ has been completed. The node $X$ sends a message to unlock entities referenced by $T_i$. The central node after receiving this message releases the locks and starts assigning locks to waiting transactions.
- *Avoid Acknowledgments, Assign Sequence Numbers.* In the centralized control algorithm, the acknowledgments are needed by the central node (or node $X$ in the above extension) to find out if the values of the write set have been written in the database at every node. But it is not necessary for the central node to wait for this to happen; it is sufficient for the central node to guarantee that the write set values are written at every node in the same order as they were performed at the central node. To achieve this the central node can assign a monotonically increasing *sequence number* to each transaction. The sequence number is appended to the write set of the transaction and is used to order the update of the new values into the database at each node.

Now the central node does not have to wait for any acknowledgments, but the equivalent effect is achieved. This can make the centralized control algorithm more efficient.

Sequence numbers may cause additional problems. Suppose two transactions $T_5$ and $T_6$ are assigned sequence numbers 5 and 6, respectively, by the central node. Let us further suppose that $T_5$ and $T_6$ have no entities in common and so do not conflict. If transaction $T_5$ is very long, transaction $T_6$, which arrived at the central node after $T_5$, may be ready to write the values of its write set, but this operation for $T_6$ must wait at all nodes for $T_5$. A simple solution to this problem is to attach the sequence numbers of all lower-numbered transactions for which a given transaction must wait before writing in the database. This list is called a *wait-for* list. In such a case, a transaction waits only for the transactions in its wait-for list. The wait-for list is attached to the write set of each transaction. In some cases the size of the wait-for list can grow very large, but transitivity among sequence numbers in wait-for lists can be used to reduce it. Moreover a complement of this wait-for list is called *do not-wait-for* list can also be used. Many such ideas are discussed in [14]. The notion of wait-for list is similar to causal ordering as discussed in [27]. In causal ordering, a message carries information about its transitive causal predecessors and the overheads to achieve this can be reduced by requiring that each message carries information about its direct predecessor only. Causal ordering has also been discussed in [7].

- *Global Two-phase Locking*. This is a simple variation of the centralized locking mechanisms. Instead of a transaction getting all locks in the beginning and releasing all locks in the end, the policy of two-phase locking is employed. Each transaction obtains the necessary locks as they are needed, computes, and then releases locks on entities that are no longer needed. A transaction cannot get a lock after it has released any lock. So if more locks are needed in the future, it should hold on to all the present locks. The other parts of the algorithm remain the same as before.
- *Primary Copy Locking*. In this variation, instead of selecting a node as the central controller, a copy of each entity on any node is designated as the primary copy of the entity. A transaction must obtain the lock on the primary copy of all entities referenced by it. At any given time the primary copy contains the most up-to-date value for that entity.

It is important to point out that locking approaches are in general *pessimistic*. For example, two-phase locking is a sufficient condition rather than the necessary condition for serializability. As an example, if an entity is only used by a single transaction, it can be locked and unlocked freely. The question is, “How can we know this?” Since this information is not known to the individual transaction, it is usually not utilized. Thus locking that is based on prevention of access does not fully benefit from actual favorable conditions that may exist.

## 2.2 Algorithms Based on Time-Stamp Mechanism

Timestamp is a mechanism in which the serialization order is selected a priori; the transaction execution is obliged to obey this order. In timestamp ordering, each transaction is assigned a unique timestamp by the scheduler or concurrency controller. Obviously, to achieve unique timestamps for transactions arriving at different nodes of a distributed system, all clocks at all nodes must be synchronized or else two identical timestamps must be resolved.

Lamport [20] has described an algorithm to synchronize distributed clocks via message passing. If a message arrives at a local node from a remote node with a higher timestamp, it is assumed that the local clock is slow or behind. The local clock is incremented to the timestamp of the recently received message. In this way all clocks are advanced until they are synchronized. In the other scheme where two identical timestamps must not be assigned to two transactions, each node assigns a timestamp to only one transaction at each tick of the clock. In addition the local clock time is stored in higher-order bits and the node identifiers are stored in the lower-order bits. Because node identifiers are different, this procedure will ensure unique timestamps.

When the operations of two transactions conflict, they are required to be processed in timestamp order. It is easy to prove that timestamp ordering (TSO) produces serializable histories. Thomas [29] has studied the correctness and implementation of this approach and described it. Essentially each node processes conflicting operations in timestamp order, each read-write conflict relation and write-write conflict relation is resolved by timestamp order. Consequently all paths in the relation are in timestamp order and, since all transactions have unique timestamps, it follows that no cycles are possible in a graph representing transaction histories.

### 2.2.1 Timestamp Ordering with Transaction Classes

In this approach, it is assumed that the read set and the write set of every transaction is known in advance. This information is used to group transactions into predefined classes. A *transaction class* is defined by a read set and a write set. A transaction *T* is a member of a class *C* if the read set of *T* is a subset of the read set of class *C* and the write set of *T* is a subset of the write set of class *C*. Class definitions are used to provide concurrency control. This mechanism was used in the development of a prototype distributed database management system called SDD-1, developed by the Computer Corporation of America [3].

### 2.2.2 Distributed Voting Algorithm

This algorithm uses distributed control to decide which transaction can be accepted and executed. The nodes of the distributed database system communicate among themselves and vote on each transaction. If a transaction gets a majority of *OK* votes, it is accepted for execution and completion. A transaction may also receive a *reject* vote, in which case it must be restarted. In addition to voting *OK* and *reject*, nodes can also defer or postpone voting on a particular transaction.

This approach is a result of the work of Thomas [29]. The timestamps are maintained for the database entities. A timestamp on an entity represents the time when this entity was last updated.

## 2.3 Algorithms Based on Rollback Mechanisms

As we have seen in the last two sections, timestamp algorithms are a major departure from the locking or the wait mechanisms. In this section, a family of nonlocking or *optimistic* concurrency control algorithms [19] are presented. In this approach, the idea is to validate a transaction against a set of previously committed transactions. If the validation fails, the read set of the transaction is updated and the transaction repeats its computation and again tries for validation. The validation phase will use conflicts among the read sets and the write sets along with certain timestamp information. The validation procedure starts when a transaction has completed its execution under the optimistic assumption that other transactions would not conflict with it. The optimistic approach maximizes the utilization of syntactic information and attempts to make use of some semantic information about each transaction. If no a priori information about an incoming transaction is available to the concurrency controller, it cannot preanalyze the transaction and try to guess potential effects on database entities. On the other hand, maximum information is available when a transaction has completed its processing. A concurrency controller can make decisions about which transaction must abort while other transactions may proceed. This decision can be made at the time of arrival of a transaction, during the execution of a transaction, or the decision can be made at the end of processing. Decisions made at arrival time will tend to be pessimistic and decisions made at the end may invalidate the transaction processing and require rollback. If the transactions' effects are kept in a private space and are not made known to other transactions until the concurrency controller ensures their correctness, one can design concurrency control mechanisms that employ maximum information at the cost of restarting some transactions. The extent of this restart will be proportional to the degree of conflict among concurrent transactions. A similar approach was suggested in [19] for a centralized hierarchical database system and was studied further in [4], [8].

There are four phases in the execution of a transaction in the optimistic concurrency control approach:

1) *Read*: Since reading a value of an entity cannot cause a loss of integrity, reads are completely unrestricted. A transaction reads the values of a set of entities (called read set) and assigns them to a set of local variables. The names of local variables have one-to-one correspondence to the names of entities in the databases, but the values of local variables are an instance of a past state of the database and are only known to the transaction. Of course since a value read by a transaction could be changed by a write of another transaction, making the read value incorrect, the read set is subject to validation. The read set is assigned a timestamp denoted by $\Pi(R_i)$.
2) *Compute*: The transaction computes a set of values for data entities called the write set. These values are assigned to a set of corresponding local variables. Thus all writes after computation take place on a transaction's copy of the entities of the database.
3) *Validate*: The transaction's local read set and write set are validated against a set of committed transactions. Details of this phase constitute a main part of this algorithm and are given in the next section.
4) *Commit and Write* (called *write* for short): If the transaction succeeds in validation, it is considered committed in the system and is assigned a timestamp denoted by $\Pi(W_i)$. Otherwise the transaction is rolled back or restarted at either the compute phase or the read phase. If a transaction succeeds in the validation phase, its write set is made global and the values of the write set become values of entities in the database at each node.

All four phases of concurrently processing transactions can be interleaved, but the read phase should precede the computation and validation phase.

### 2.3.1 The Validation Phase

The concurrency controller can utilize syntactic information, semantic information, or a combination of the two. Here we discuss the use of syntactic information in the context of a validation at one node only. For use of semantic information, we refer the reader to [5].

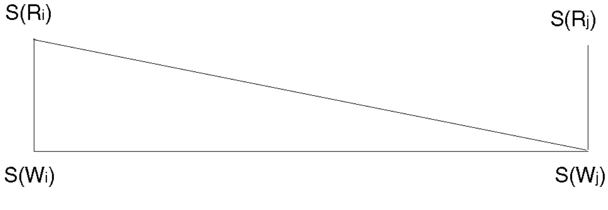
Kung and Papadimitriou [18] have shown that when only the syntactic information is available to the concurrency controller, serializability is the best achievable correctness criterion. We now describe the validation phase.

A transaction enters the validation phase only after completing its computation phase. The transaction that enters the validation phase before any other transaction is automatically validated and committed. This is because initially the set of committed transactions is empty. This transaction writes updated values in the database. Since this transaction may be required to validate against future transactions, a copy of its read and write sets is kept by the system. Any transaction that enters the validation phase validates against the set of committed transactions that were concurrent with it. As an extension the validation procedure could include validation against other transactions currently in the validation phase.

Consider two transactions $T_i$ and $T_j$. Let $S(R_i)$ and $S(R_j)$ be the read sets and $S(W_i)$ and $S(W_j)$ be the write sets of $T_i$ and $T_j$, respectively. Let $\Pi(R_i)$ and $\Pi(R_j)$ denote the time when the last item of the read set $S(R_i)$ and $S(R_j)$ were read from the database and let $\Pi(W_i)$ and $\Pi(W_j)$ denote the time when the first item of the write set $S(W_i)$ and $S(W_j)$ will be or were written in the database.

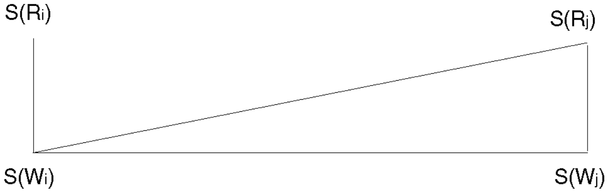
Assume $T_i$ arrives in the system before $T_j$. Let $T_j$ be a committed transaction when the transaction $T_i$ arrives for validation. Now there are four possibilities:

1) If $T_i$ and $T_j$ do not conflict, $T_i$ is successful in the validation phase and can either proceed or follow $T_j$.
2) If $S(R_i) \cap S(W_j) \neq \emptyset$ and $S(R_j) \cap S(W_i) \neq \emptyset$, $T_i$ fails in the validation phase and restarts.
3) If $S(R_i) \cap S(W_j) \neq \emptyset$ and $S(R_j) \cap S(W_i) = \emptyset$, $T_i$ is successful in validation. $T_i$ must proceed $T_j$ in any serial history since $\Pi(R_i) < \Pi(W_j)$. This possibility is illustrated as follows:



The edge between $S(W_i)$ and $S(W_j)$ does not matter because if $S(W_i)$ intersects with $S(W_j)$, then $S(W_i)$ can be replaced by $S(W_i) - [S(W_i) \cap S(W_j)]$. In other words, $T_i$ will write values for only those entities that are not common with the write set of $T_j$. If we do so, we get the equivalent effect as if $T_i$ were written before $T_j$.

4) If $S(R_i) \cap S(W_j) = \emptyset$ and $S(R_j) \cap S(W_i) \neq \emptyset$, $T_i$ is successful in validation. $T_i$ must follow $T_j$ in any serial history since $\Pi(W_i) > \Pi(R_j)$. This possibility is illustrated as follows:



For a set of concurrent transactions we proceed as follows: For each transaction that is validated and enters the list of committed transactions, we draw a directed edge according to the following rules:

- If $T_i$ and $T_j$ do not conflict, do not draw any edge.
- If $T_i$ must precede $T_j$, draw an edge from $T_i$ to $T_j$, $T_i \rightarrow T_j$.
- If $T_i$ must follow $T_j$, draw an edge from $T_j$ to $T_i$, $T_i \leftarrow T_j$.

Thus, a directed graph is created for all committed transactions with transactions as nodes and edges as explained above.

When a new transaction $T_i$ arrives for validation, it is checked against each committed transaction to check if $T_i$ should precede or follow, or if the order does not matter.

*Condition for Validation*: There is never a cycle in the graph of committed transactions because they are serializable. If the *validating transaction* creates a cycle in the graph, it must restart or rollback. Otherwise, it is included in the set of committed transactions. We assume the validation of a transaction to be in the critical section so that the set of committed transactions does not change while a transaction is actively validating.

In case a transaction fails in validation, the concurrency controller can restart the transaction from the beginning of the read phase. This is because the failure in the validation makes the read set of the failed transaction incorrect. The read set of such a transaction becomes incorrect because of some write sets of the committed transactions. Since the write sets of the committed transactions meet the read set of the failed transaction (during validation), it may be possible to update the values of the read set of the transaction at the time of validation. If this is possible, the failed transaction can start at the beginning of the compute phase rather than at the beginning of the read phase. This will save the I/O access required to update the read set of the failed transaction.

### 2.3.2 Implementation Issues

Let $T_i$ be the validating transaction and let $T_j$ be a member of a set of committed transactions.

The read sets and write sets of the committed transactions are kept in the system. The transaction is validated against the committed transactions. The committed transactions are selected in the order of their commitment. The read set is updated by the conflicting write set at the time of each validation. If none of the transactions conflict with the validating transaction, it is considered to have succeeded in the validation and hence to have committed.

This obviously requires updating a given entity of the read set many times and thus is inefficient. But one nice property of this procedure is that the transaction does not have to restart from the beginning and does not have to read the database on secondary storage.

A practical question is whether the read sets and write sets can be stored in memory. The transactions $T_j$ that must be stored in memory must satisfy the following condition: If $T_j$ is a committed transaction, store $T_j$ for future validation if $T_i \notin$ set of committed transaction such that:

$$\{\Pi(R_i) < \Pi(W_j)\} \text{ AND } \{S(R_i) \cap S(W_j) \neq \phi\}$$

It has been shown that the set of transactions to be stored for future validation will usually be small [4]. In general, a maximum size of the number of committed transactions that can be stored in the memory can be determined at design time. In case the number of committed transactions exceed this limit, the earliest committed transaction $T_j$ can be deleted from this list. But care should be taken to restart (or invalidate) all active transactions $T_i$ for which $\Pi(R_i) < \Pi(W_j)$ before $T_j$ is deleted.

A DETAILED EXAMPLE. This example illustrates a variety of ideas of optimistic approach and its advantages over locking. We assume that a history is presented to a scheduler (concurrency controller). The scheduler either accepts or rejects the history. It does so by trying conflict preserving exchanges on the input history to check if it can be serializable.

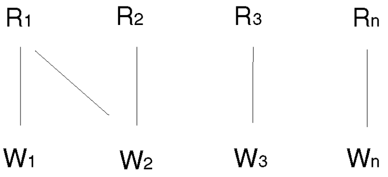
In this example, we use $R_i$ and $W_i$ to represent the read action (as well as the read set) and the write action (as well as the write set) of a transaction $T_i$.

Let $h$ be an input history of $n$ transactions to the scheduler as follows:

$$h = R_1 R_2 W_2 R_3 W_3 \cdots\cdots\cdots R_n W_n W_1$$

Here transaction $T_1$ executes the read actions, followed by the read / write action of $T_2$, $T_3$, ..., $T_n$, followed by the write actions of $T_1$.

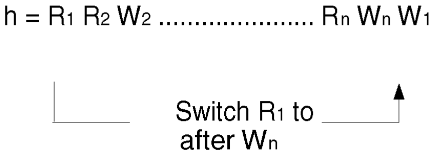
Suppose $R_1$ and $W_2$ conflict as represented by an edge as follows:



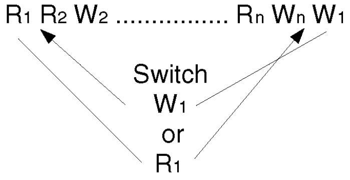
The history $h$ is not allowed in the locking protocols because $W_2$ is blocked by $R_1$. If $T_1$ is a long transaction and $T_2$ is a small transaction, the response time for $T_2$ will suffer. In general $T_1$ can block $T_2$, $T_2$ can block $T_3$ (if $T_2$ and $T_3$ have a conflict) and so on.

Let us consider several cases in optimistic approach.

**Case 1**: For the history $h$, in the optimistic approach of Kung and Robinson [19], $T_i$ ($i = 2, \ldots, n$) can commit. Write sets ($W_i$ s) of committed transactions are saved to validate against the read set of $T_1$. Basically the conflict preserving exchange (switch) as follows is attempted so that $R_1$ can be brought next to $W_1$.



**Case 2**: An extension of this idea is to try the either exchange (switch) as follows:



The resulting histories can be either

$$R_1 W_1 R_2 W_2 \cdots\cdots\cdots R_n W_n$$

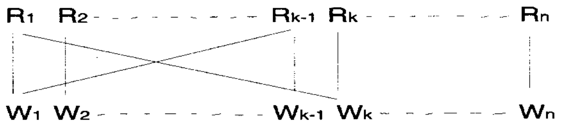
or

$$R_2 W_2 \cdots\cdots\cdots R_n W_n R_1 W_1.$$

For switching $W_1$, we would need to save not only the write sets of committed transactions, but also the read sets of committed transactions. This will allow more histories to be acceptable to the scheduler.

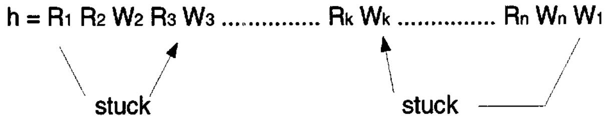
**Case 3**: A further extension of this idea is to try switching $R_1$ toward $W_1$ and $W_1$ toward $R_1$ if conflict preserving exchanges are possible.

Consider the conflict as follows:


R1 R2 - - - - - - - - - Rk-1 Rk - - - - - - - Rn
W1 W2 - - - - - - - - - Wk-1 Wk - - - - - - - Wn

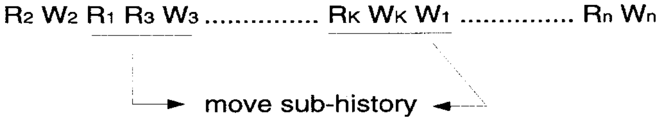
Consider the history $h = R_1 R_2 W_2 \cdots R_{K-1} W_{k-1} R_K W_K \cdots R_n W_n W_1$. Because of a conflict edge between $R_1$ and $W_K$, $R_1$ can be scheduled only before $W_K$. Similarly due to the conflict edge $R_{K-1}$ and $W_1$, $W_1$ can be scheduled only after $R_{K-1}$. Switching $R_1$ and $W_1$, the scheduler can get a serializable history $R_2 W_2 \cdots R_{K-1} W_{K-1} R_1 W_1 R_K W_K \cdots R_n W_n$. Using the switching of $R_1$ or $W_1$ alone would not have allowed this history to be acceptable to a scheduler.

Finally we consider the case where both $R_1$ and $W_1$ are stuck due to conflicts. Consider the history:


h = R1 R2 W2 R3 W3 ............... Rk Wk ............... Rn Wn W1
stuck
stuck

$R_1$ can switch up to $T_3$ and $W_1$ can switch up to $T_K$ due to conflicts (say $R_1 W_3$ and $W_1 W_K$).

The scheduler can try to move the subhistory $R_1 R_3 W_3$ to the right and $R_K W_K W_1$ to the left as shown next:


R2 W2 R1 R3 W3 ............... RK WK W1 ............... Rn Wn
move sub-history

We can get a history as follows.

$$R_2 W_2 R_K W_K R_1 W_1 R_3 W_3 \cdots\cdots\cdots R_n W_n$$

which is serializable.

### 2.3.3 Implementation of Validations

Let us now illustrate how these ideas for optimistic can be implemented. Consider $n$ transactions. Assume $T_1$ starts before $T_2$, but finishes after $T_n$. Let $T_2 T_3 \cdots T_n$ finish in order.

Since $T_2$ is the first transaction to validate, it is committed automatically. So we have a conflict graph with a node for $T_2$ as follows:

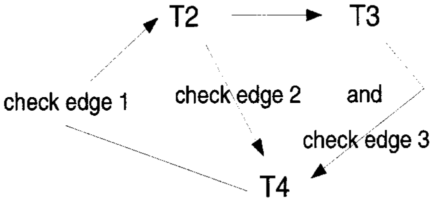
$$\mathbf{T_2}$$

When $T_3$ arrives for validation, the read set of $T_3$ is validated against write set of $T_2$. If they conflict, the edge $T_3 \rightarrow T_2$ is drawn. Next, the write set of $T_3$ is validated against the read set of $T_2$ leading to the edge $T_2 \rightarrow T_3$. Since this causes a cycle, $T_3$ is aborted. Otherwise, $T_3$ is serialized with $T_2$. So we have a conflict graph say as follows:

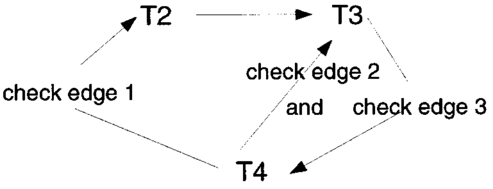
$$\mathbf{T_2 \rightarrow T_3}$$

For a transaction $T_4$ the edges are checked as follows:

Check the edge $T_4 \rightarrow T_2$. If it exists, check the edge $T_2 \rightarrow T_4$. Abort if both edges exist. If only $T_4 \rightarrow T_2$ exists, do not check the edge $T_4 \rightarrow T_3$, but check the edge $T_3 \rightarrow T_4$ only. This requires checking only three edges as follows:



If edge $T_4 \rightarrow T_2$ does not exist, there is no need to check $T_2 \rightarrow T_4$. In this case check the edge $T_4 \rightarrow T_3$ and $T_3 \rightarrow T_4$. Once again only three edges are checked as follows:



So, in general, for every new transaction that comes for validation, only $n$ edges are checked if there are $n - 1$ committed transactions. This may be more efficient implementation than checking for a cycle for the conflict graph of $n$ transactions for each validation.

Now we present a theorem that relates the rollback of optimistic with deadlock of locking approach:

THEOREM 1 [8]. *In a two step transaction model (all reads for a transaction precede all writes) whenever there is a transaction rollback in the optimistic approach due to a failure in the validation, there will be a deadlock in the locking approach (unless deadlocks are not allowed to occur) and will cause a transaction rollback.*

PROOF. For deadlock detection, the system can produce a wait-for digraph in which the vertices represent the transactions active in the system. An edge between two transactions in the wait-for graph is drawn if and only if one transaction holds a read-lock or a write-lock and the other transaction is requesting a write-lock on the same item. This will happen when the read-set or the write-set of the first transaction conflicts (intersects) with the write-set of the second transaction. An edge in the dynamic conflict graph exists in exactly the same case.

Thus a wait-for graph has the same vertices (i.e., the set of all active transactions) as the dynamic conflict graph and the edges in the wait-for graph correspond one to one with the edges in the dynamic conflict graph. Hence the wait-for graph is identical to the dynamic conflict graph and a cycle in the wait-for graph occurs whenever there is a cycle in the dynamic conflict graph. A deadlock occurs when there is a cycle in the wait-for graph and to resolve the deadlock, some transaction must be rolled back. Since validation of a transaction fails and a rollback happens when there is a cycle in the dynamic conflict graph, the assertion of the theorem is concluded. □

## 3 PERFORMANCE EVALUATION OF CONCURRENCY CONTROL ALGORITHM

There are two main criteria for evaluating the performance of core control algorithms. We discuss them in some detail as follows:

### 3.1 Degree of Concurrency

This is the set of histories that are acceptable to a scheduler. For example a serial history has the lowest degree of concurrency. 2PL and optimistic approaches provide a higher degree of concurrency. The concurrency control algorithms have been classified in various classes based on the degree of concurrency provided by them in [24]. The concurrency control algorithms for distributed database processing have been classified in [7]. We specifically point out the classes of global two-phase locking (G2PL) and local two-phase locking (L2PL). All histories in class G2PL are characterized by global lock points. Since each node is capable of independent processing, the global history can be serializable if each node maintains the same order of lock points for all conflicting transactions locally. The class L2PL contains the class G2PL and provides a higher degree of concurrency [7]. In a history for the class DSTO (distributed serializable in the time stamp order), the transactions are guaranteed to follow in the final equivalent serial history, the same order as the transaction's initial access or event $\alpha$. $\alpha$, $\omega$ events are discussed in Appendix A.

In contrast, the class DSS (distributed strict serializability) the histories retain the completion order of transactions based on the event $\omega$. The class DSTO is contained in class DSS. Finally, the class DCP (distributed conflict preserving) is based on the notion the a read or write action is freely rearranged as long as the order of conflicting accesses is preserved. The serializability is guaranteed by maintaining a acyclic conflict graph that is constructed for each history.

#### 3.1.1 The Hierarchy

All the classes G2PL, L2PL, DCP, DSTO, and DSS are serializable and form a hierarchy based on the degree of concurrency. Fig. 3 depicts the hierarchy, where SR is the set of all serializable histories.

In Fig. 3, each possible intersection of these classes is marked by '.i' where i is from 1 to 11, and the exemplary history for area '.1' is denoted as 'h.i'. Some of the histories are composite (formed by concatenating two histories). The transaction set and conflict information are given below.

Let there be two nodes represented by $N = \{1, 2\}$, and seven transactions denoted by $T = \{a, b, c, d, e, f, g\}$.

The atomic operations for each transaction are as shown below:

$$\text{Trans. `a'} = \left\{R_a^1[x],\ W_a^1[y],\ W_a^2[y]\right\};$$

$$\text{Trans. `b'} = \left\{R_b^1[w],\ W_b^1[y],\ W_b^2[y]\right\};$$

$$\text{Trans. `c'} = \left\{R_c^2[u],\ W_c^1[v],\ W_c^2[v]\right\};$$

$$\text{Trans. `d'} = \left\{R_d^2[v],\ W_d^1[v],\ W_d^2[v]\right\};$$

$$\text{Trans. `e'} = \left\{R_e^1[w],\ W_e^1[v],\ W_e^2[v]\right\};$$

$$\text{Trans. `f'} = \left\{R_f^2[t],\ W_f^1[w],\ W_f^2[w]\right\};$$

$$\text{Trans. `g'} = \left\{R_g^2[w],\ W_g^1[y],\ W_g^2[y]\right\};$$

Basically, each transaction reads an entity and broadcasts the update to both nodes.

The hierarchical relation among the classes DCP, DSS, and G2PL is similar to that in [24]. However, the classes L2PL and DSTO and their relationships with other classes is different. Note that, unlike the class 2PL in the centralized database system, the class L2PL which also uses two-phase locking but with local freedom of choosing the lock points is not contained in DSS.

h.1: $R_b^1[w]W_b^2[y]R_c^2[u]W_c^1[v]W_c^2[v]W_b^1[y]$

h.2: $R_b^1[w]R_a^1[x]W_a^2[y]W_a^1[y]W_b^1[y]W_b^2[y]$

h.3: $R_a^1[x]R_b^1[w]W_a^1[y]W_b^1[y]W_a^2[y]W_b^2[y]$

h.4: h.2 + h.3

h.5: $R_a^1[x]R_b^1[w]R_f^2[t]W_f^1[w]W_a^1[y]W_f^2[w]W_a^2[y]$
$W_b^2[y]W_b^1[y]$

h.6: h.2 + h.5

h.7: $R_e^1[w]R_f^2[t]W_f^1[w]W_f^2[w]R_d^2[v]W_d^2[v]W_d^1[v]$
$W_e^2[v]W_e^1[v]$

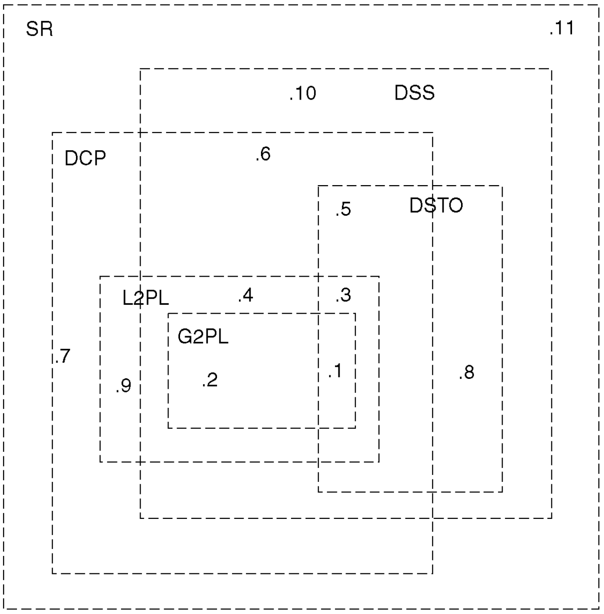


Fig. 3. The hierarchy of the classes SR, DCP, L2PL, G2PL, DSTO, and DSS.

h.8: $R_c^2[u]R_e^1[w]W_e^1[v]W_e^2[v]R_d^2[v]W_c^1[v]W_c^2[v]$
$W_d^1[v]W_d^2[v]$

h.9: $R_f^2[t]W_f^2[w]R_g^2[w]W_g^2[y]R_e^1[w]W_g^1[y]W_e^2[v]$
$W_e^1[v]W_f^1[w]$

h.10: h.6 + h.8

h.11: h.7 + h.10

## 3.2 System Behavior

One can evaluate the performance of a concurrency control algorithms by studying the response time for transactions, throughput of transactions per second, rollback or blocking of transactions, etc. Such measures are system dependent and change as technology changes. Several research papers have done simulation, analytical, and experimental study of a wide range of algorithms. These studies tend to identify the conditions under which a specific approach will perform better. For example, in [4], we have shown after detailed simulations that the optimistic approach performs better than locking when there is a mix of large and small transactions. This is contrary to the wisdom that optimistic performs better when there are few conflicts. We found that in the case of low conflicts, in optimistic approach there are fewer aborts, but in locking there is less blocking. Similarly, if a lot of conflicts occur, both locking and optimistic algorithms suffer. Thus, one could conclude that if the cost of rollback and validation is not considerably high, in both locking and optimistic, the transactions will either suffer or succeed.

In many applications, it has been found that conflicts are rare [12], [16], [17]. We present another strawman analysis. Assume that the database size is M and the read set and write set size is B. $C_B^M$ represents the number of combinations for choosing $B$ objects from a set of $M$ objects.

The probability that two transactions do not share a data object is given by the following term:

$$\frac{C_B^M \ * \ C_B^{(M-B)}}{C_B^M \ * \ C_B^M}$$

This term is equal to

$$\left(\frac{M-B}{M}\right) * \left(\frac{M-B-1}{M-1}\right) * \cdots * \left(\frac{M-2B+1}{M-B+1}\right)$$

Lower bound on this term

$$= \left(\frac{M-2B+1}{M-B+1}\right)^B .$$

Maximum probability that two transactions will share a data object is given by

$$1 - \left(\frac{M-2B+1}{M-B+1}\right)^B .$$

By plugging some sample values for $B$ and $M$, we get the following:

| B | M | Probability of Conflict P(C) |
|---|---|---|
| 5 | 100 | 0.0576 |
| 10 | 500 | 0.0025 |
| 20 | 1,000 | 0.1130 |

The probability of a cyclic conflict is order $(P(C))^2$ which is quite small.

We have conducted a simulation study [4] that illustrates the issues of arrival rate, relates multiprogramming level, frequency of cycles in a database environment. In Fig. 4, we show that the degree of multiprogramming is low for a variety of transaction arrival rates in a sample database.

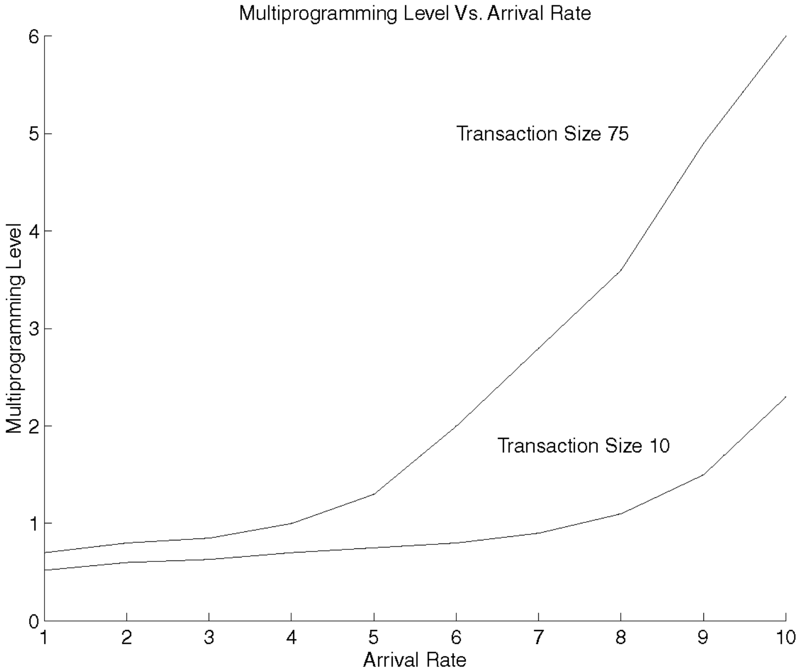


Fig. 4. Database size = 100, I/O cost = 0.025, CPU cost = 0.0001.

In Fig. 5, we show that the probability of a cycle is quite low for low degrees of multiprogramming.

In Fig. 6, we found that optimistic performs better than locking for very low arrival rates. Details of this study can be found in [4].

## 4 More Ideas for Increasing Concurrency

### 4.1 Multidimensional Time Stamps

There are several variations of timestamp ordering. For example, multiple versions [25] of item values have been used to increase the degree of concurrency. The conventional time stamp ordering tends to prematurely determine the serializability order, which may not fit in with the subsequent history, forcing some transactions to abort. The multidimensional time stamp protocol [21] provides a higher degree of concurrency than single time stamp algorithms. This protocol allows the transaction to have a time stamp vector of up to *k* elements. The maximum value of *k* is limited by twice the maximum number of operations in a single transaction. Each operation may set up a new dependency relationship between two transactions. The relationship (or order) is encoded by making one vector less than another. A single time stamp element is used to bear this information. Earlier assigned elements are more significant in the sense that subsequent dependency relationships cannot conflict with previously encoded relationships. Thus the scheduler can decide to accept or abort an operation based on the dependency information derived from all proceeding operations. In other words, the scheduler can use the approach of dynamic timestamp vector generations for each transaction and dynamic validation of conflicting one can use the approach of dynamic timestamp vector generations for each transaction and dynamic validation of conflicting transactions to increase the degree of concurrency. The class of multidimensional time stamp vectors intersects with the class SSR and 2PL and is contained in the class DSR. Classes 2PL, SSR, and DSR are defined as in [24].

### 4.2 Relaxations of Two-Phase Locking

In [22], we have provided a clarification of the definition of two-phase blocking. A restricted non two-phase locking (RN2PL) class that contains the class of 2PL has been formally defined. An interesting interpretation of the RN2PL is given as follows.

A transaction (a leaser) may release a lock (rent out a lock token) before it may still request some more locks. If a later transaction (a leasee) subsequently obtains such a released lock (rents the lock token), it cannot release this lock (sublease the lock token) until ALL its leasers will not request any more locks. (Now the leasers are ready to transfer all their lock tokens to leasees. So, each of their leasees can be a new leaser.)

This scenario enforces acyclic leaser-leasee relationships, and thus produces only serializable histories. Further, the locking sequence may not be two-phased. It is not appropriate to claim that either protocol is superior to the other because many conditions need to be considered for such a comparison. Since two-phase locking is a special case of restricted-non-two-phase locking, it gives the flexibility for some transactions to be non-two-phase locked. In some cases, it would be desirable to allow long-lived transactions to be non-two-phase locked to increase the availability of data items.

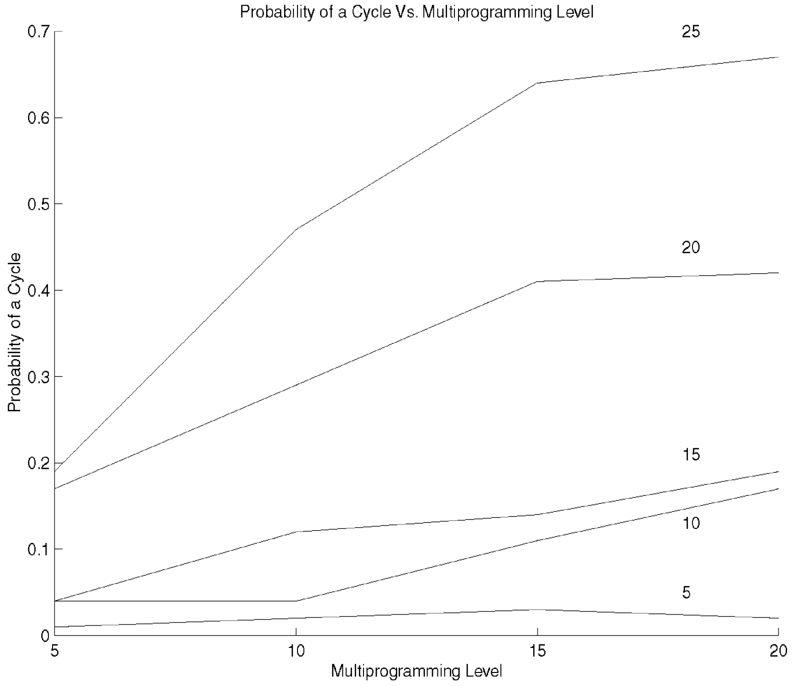


Fig. 5. Database size = 500, transaction size = 5, 10, 15, 20, 25.

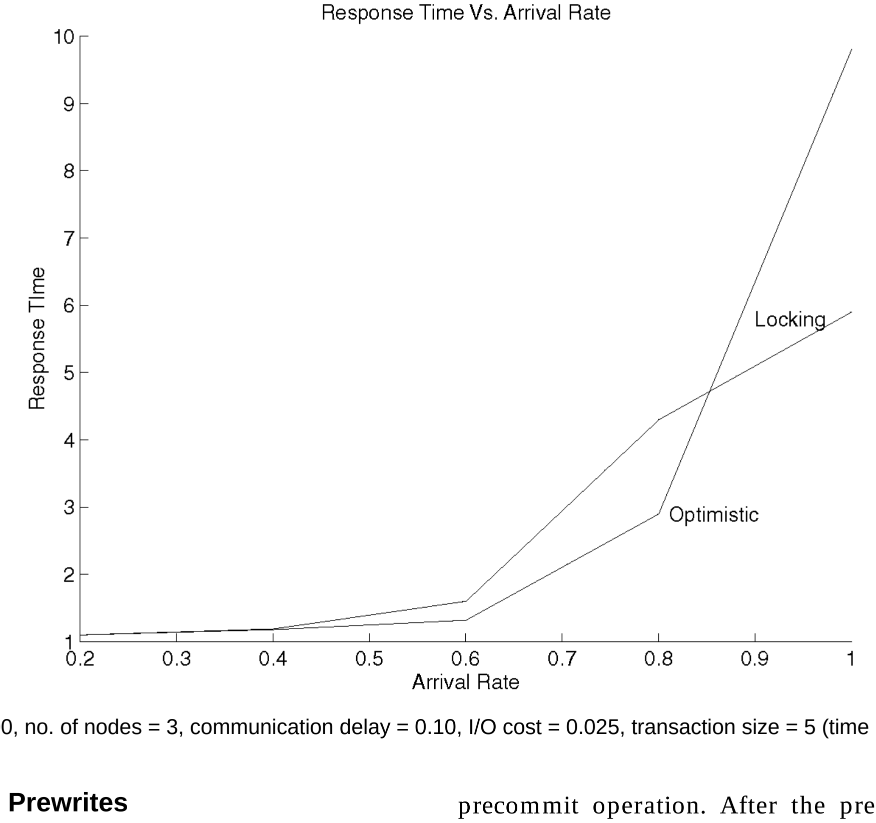


Fig. 6. Database size = 200, no. of nodes = 3, communication delay = 0.10, I/O cost = 0.025, transaction size = 5 (time in seconds).

### 4.3 System Defined Prewrites

In [23], we have introduced a prewrite operation before an actual write operation is performed on database files. A prewrite operation announces the value that a transaction intends to write in future. A prewrite operation does not change the state of the data object. Once all the prewrites of a transaction are announced, the transaction executes a precommit operation. After the precommit, another read transaction is permitted to read the announced prewrite values even before the other transaction has finally updated the data objects and committed. The eventual updating on stable storage may take a long time. This allows nonstrict executions and increases the potential concurrency as compared to the algorithms that permit only read and write operations on the database files. A user does not explicitly mention a prewrite operation but the system introduces a prewrite operation before every write.

Short duration transactions can read the value of a data item produced but not yet released by a long transaction before its commit. Therefore, using prewrites, one can balance a system consisting of short and long transactions without causing delay for short duration transactions.

### 4.4 Flexible Transactions

The flexible transaction model [32] supports flexible execution control flow by specifying two types of dependencies among the subtransactions of a global distributed transaction:

1) execution ordering dependencies between two subtransactions, and
2) alternative dependencies between two subsets of subtransactions.

A flexible transaction allows for the specification of multiple alternate subsets of subtransactions to be executed and results in the successful execution and commitment of the subtransactions in *one* of those alternate subsets, the execution of a flexible transaction can proceed in several different ways. The subtransaction in different alternate subsets may be attempted simultaneously, as long as any attempted subtransactions not in the committed subset of subtransactions can either be aborted or have their effects undone. The flexible transaction model increases the failure resilience of global transactions. In [32], we have defined a weaker form of atomicity, termed *semiatomicity*, that is applicable to flexible transactions. Semiatomicity allows a flexible transaction to commit as long as a subset of its subtransactions that can represent the execution of the entire flexible transaction commit. Semiatomicity enlarges the class of executable global transactions in a heterogeneous distributed database system.

### 4.5 Adaptable Concurrency Control

Existing database systems can be interconnected resulting in a heterogeneous distributed database system. Each site in such a system could use a different strategy for concurrency control. For example, one site could be using the two-phase locking concurrency control method while another could be running the optimistic method. Since it may not be possible to convert such different systems and algorithms to a homogeneous system, solutions must be found to deal with such heterogeneity. Already research has been done toward the designing of algorithms for performing concurrent updates in a heterogeneous environment [31]. The issues of global serializability and deadlock resolution have been solved. The approach in [11] is a variation of the optimistic concurrency control for global transactions while allowing individual sites to maintain their autonomy.

Another concept that has been studied in the Reliable, Adaptable, Interoperable Distributed (RAID) database system [10] involves facilities to switch concurrency control methods. A formal model for an *adaptable* concurrency control [11] suggested three approaches for dealing with various system and transaction's states: generic state, converting state, and suffix sufficient state. The generic state method requires the development of a common data structure for all the ways to implement a particular concurrency controller (called sequencer). The converting state method works by invoking a conversion routine to change the state information as required by a different method. The suffix sufficient method requires switching from one method to another by overlapping the execution of both methods until certain termination conditions are satisfied.

## 5 Conclusions

Concurrency Control is a problem that arises when multiple processes are involved in any part of the system. Earlier ideas of notions of serializability and the concept of two-phase locking were discussed in [13]. The ideas of time stamps were introduced by [29]. The optimistic approach was proposed by [19]. The classes of serializability and the formalism for concurrency control was presented in [24]. Several books that detail these subjects have been published [6], [25], [2], in addition to survey papers [1], [5]. The performance evaluation was studied in [14]. In most commercial systems, the most popular mechanism for concurrency control is two-phase locking [17]. The ideas of adaptable concurrency control were published in [11] and were implemented in the RAID system [10]. It has been commented by system experts that concurrency control only contributes 5 percent to the response time of a transaction and so even a simple two-phase locking protocol should suffice. However, due to the many interesting ideas that came into play in distributed database systems in the context of replication and reliability, research in concurrency control is continuing. Some studies are being done for object-oriented systems while others are dealing with semantics of transactions and weaker form of consistency. Over a hundred Ph.D. thesis that study some aspect of concurrency control have been produced.

We continue to learn of new ideas such as flexible transactions, value-dates, prewrites, degrees of commitment and view serializability [9]. In large scale systems, it is difficult to block access to database objects for transactions. If a system has to perform 10,000 transactions per second, the locking as we know today will not be a solution. We suggest readers to learn from variety of books that are available on both theory and implementation of concurrency control mechanisms.

## Appendix A
## Basic Terminology

A *distributed database management system* (DDBMS) is a database system distributed among a set of *nodes N* connected by communication links. Each node has its own independent computing resources.

The database is modeled by a set of *logical database entities* which may have one or more *physical copies of data value*. The database entities are accessed by unique names; how this naming is maintained is insignificant to this paper. The database may be either completely or partially replicated, or it may be partitioned on different nodes.

A distributed database is *consistent* if it satisfies some predefined assertions about the intrinsic characteristics of the data values. For a replicated distributed database, it is necessary for the physical copies of the same database entity on different nodes to remain identical.

The user actions on a distributed database consists of a sequence of atomic operations. An *atomic operation* is represented by $\sigma_i = A_i^j[x]$, where $i$ is a unique identification for a transaction, $j$ is a unique identification for a node, $A$ is either $R$ or $W$ representing read or write operation, and $x$ is one or more logical database entities. As far as the DDBMS is concerned, these read / write operations constitute indivisible (or atomic) operations to the database. The atomic operations are grouped into logical units called *transactions* that will preserve the database consistency if executed alone. A transaction can be viewed as a quantum change for the database from one consistent state to another; however, the consistency assertions may be temporarily violated during the execution of a transaction but must be satisfied when there are no incomplete transactions or the system is quiescent. The purpose of the concurrency control is to guarantee that the concurrent execution of a set of transactions does not result in an inconsistent database state.

The *transaction set T* represents all user transactions, and the *atomic operation set*. A transaction has to read only one copy of a replicated data entity but has to update all copies.

Two atomic operations $\sigma_i$, $\sigma_j$ *conflict* if:

1) they belong to different transactions;
2) both access the same database entity at the same node;
3) at least one of them is a write operation.

In particular, conflicting atomic operations $\sigma_i$ and $\sigma_j$ have:

1) *WR-conflict* if $\sigma_i$ is a write operation and $\sigma_j$ is a read operation;
2) *RW-conflict* if $\sigma_i$ is a read operation and $\sigma_j$ is a write operation;
3) *WW-conflict* if both $\sigma_i$ and $\sigma_j$ are write operations.

There are two special atomic operations in a transaction that are important. The *last new atomic operation* $\omega_i$ of transaction $i$ is its last atomic operation such that the access is to a new database entity or the access is at a higher level[1] than before for a previously accessed entity. Every atomic operation after $\omega_i$ either accesses some used entity or repeats a lower level access. The *earliest new atomic operation* $\alpha_i$ for a transaction $i$ is the first atomic operation which starts accessing new entities. Since each atomic operation accesses some database entities, $\alpha_i$ is simply the first atomic operation in a transaction.

For example, $\omega_i$ of the following transaction

$$R_i^1[x]W_i^2[y]W_i^2[z]W_i^3[y]W_i^3[z]$$

is $W_i^2[z]$ since $z$ is the last new entity being accessed. The *omega*$_i$ of the following transaction

$$R_i^1[x]W_i^2[y]W_i^2[z]W_i^3[y]W_i^3[z]W_i^1[x]R_i^2[z]R_i^1[x]$$

is $W_i^1[x]$ since it is the latest higher level access to any entity ($x$ in this case).

The concurrent activities of a distributed database system can be modeled as a sequence of all atomic operations. This sequence is called the *history* of the system, and is represented by a quadruple $h = < D, T, \Sigma, \pi >$, where $D$ is a distributed database, $T$ is the transaction set, $\Sigma$ is the atomic operation set, and $\pi$ is a *permutation function* which gives the permutation indices for atomic operations $\sigma$ in $h(\sigma \in \Sigma)$. For example, if a history $h$ is the following sequence

$$\alpha\beta\gamma \cdots \omega$$

then $\pi(\alpha) = 1$, $\pi(\beta) = 2, \ldots, \pi(\omega) = |\Sigma|$. A *serial history* is one in which each transaction runs to completion before the next one starts. In other words in a serial history the atomic operations of different transactions are not interleaved.

Although the system's activities can be modeled as a string of atomic operations, the activity at one node is potentially independent of those at other nodes. Each node records its own history. To capture this notion of local activities, *the node projection* $h^j = <h, \Sigma^j, \pi^j>$ of a history $h$ is defined as the subsequence of $h$ containing only those operations pertaining to node $j$ where $\Sigma^j = \{\sigma | \sigma \in \Sigma$ and $\sigma$ is performed at node $j\}$ is a subset of $\Sigma$, and $\pi^j$ is the permutation function for $h^j$, i.e., $\pi^j(\sigma_1) < \pi^j(\sigma_2)$ iff $\pi(\sigma_1) < \pi(\sigma_2)$ for $\sigma_1, \sigma_2 \in \Sigma^j$. The order of the atomic operations in h is retained in $\pi^j$.

The activities of a transaction in a distributed database system can be modeled by a sequence of operations on the database related to this transaction. This sequence is called *the transaction projection* $h_i = <h, \Sigma_i, \pi_i>$, where $\Sigma_i = \{\sigma | \sigma \in \Sigma$ and $\sigma$ belongs to transaction $i\}$ is a subset of $\Sigma$, and $\pi_i$ is the permutation function for $h_i$, i.e., $\pi_i(\sigma_1) < \pi_i(\sigma_2)$ iff $\pi(\sigma_1) < \pi(\sigma_2)$ for every $\sigma_1, \sigma_2 \in \Sigma_i$.

From the above definitions, it is clear that a serial history h has the form

$$h_{i_1}h_{i_2}h_{i_3} \cdots h_{i_t} \text{ for } i_k \in T, k = 1, \ldots, t$$

where $t = |T|$ and $i_1 i_2 i_3 \cdots i_t$ is a permutation of transaction id's ($h_{i_1}$, say, is the transaction projection for transaction $i_1$ from the serial history $h$). Note that each node projection of a serial history is essentially a sequential execution of transactions following the same permutation order $i_1 i_2 \cdots i_t$ of the serial history.

Each operation in a history transforms one database state into another one. Two histories are *equivalent* or indistinguishable if they transform a given initial state to the same final database state. The notation $\equiv$ denotes the equivalence relation between histories. A history $h$ is *serializable* iff there exists a serial history $g$ such that $h^j \equiv g^j$ for every node $j$.

If every transaction when executed alone preserves the database consistency then each node projection of a serializable history will also preserve the consistency. Since a serial history produces node projections with the same serial transaction order, a serializable history necessarily

[1] For the transaction model used here, the read operation is considered a lower level access when compared to the write operation.

generates a consistent database. An algorithm is considered correct if all its allowed histories are serializable.

The use of serializability as a correctness criterion is popular among researchers. Although nonserializable histories can be consistent when semantic information is available. we still consider serializability to be the correctness criterion. It has been shown in [18] that concurrency control algorithms with only syntactic information can at best produce serializable histories.

## ACKNOWLEDGMENTS

I have learned a lot from the research of Professors Gray, Bernstein / Goodman, Papadimitriou, Kung / Robinson, Stonebraker, and systems such as Ingres, System R, SDD-1, and RAID (Reliable, Adaptable, Interoperable Distributed) database system. I thank the designers of these systems for giving us an interesting perspective. In 1982, Professor C.V. Ramamoorthy gave me encouragement and many ideas, at the early stages of my research in concurrency control. I thank Professor Farokh Bastani, editor-in-chief of *IEEE Transactions on Knowledge and Data Engineering*, for encouraging me to write this paper for the special issue in honor of Professor C.V. Ramamoorthy.

## REFERENCES

[1] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys*, vol. 13, no. 2, pp. 185-221, 1981.

[2] P.A. Bernstein, N. Goodman, and V. Hadzilacos, *Concurrency Control and Recovery in Database Systems*, Addison-Wesley, Reading, Mass., 1986.

[3] P.A. Bernstein, D.W. Shipman, and J.B. Rothnie Jr., "Concurrency Control in a System for Distributed Databases (SDD-1)," *Trans. Database Systems*, vol. 5, no. 1, pp. 18-51, ACM, 1980.

[4] B. Bhargava, "Performance Evaluation of the Optimistic Concurrency Control Approach to Distributed Database Systems and Its Comparison with Locking," *Proc. IEEE Int'l Conf. Distributed Computing Systems*, pp. 508-517, Miami, 1982.

[5] B. Bhargava, "Concurrency Control and Reliability in Distributed Database System," *Software Eng. Handbook*, Van Nostrand Reinhold, pp. 331-358, 1983.

[6] "Concurrency Control and Reliability in Distributed Systems," B. Bhargava, ed., Van Nostrand and Reinhold, 1987.

[7] B. Bhargava and C. Hua, "A Causal Model for Analyzing Distributed Concurrency Control Algorithms," *IEEE Trans. Software Eng.*, vol. 9, pp. 470-486, 1983.

[8] B. Bhargava, "Resilient Concurrency Control in Distributed Database Systems," *IEEE Trans. Reliability*, vol. 31, no. 5, pp. 437-443, 1984.

[9] B. Bhargava, "Transaction Processing and Consistency Control of Replicated Copies During Failures," *J. Management Information Systems*, vol. 4, no. 2, pp. 93-112, 1987.

[10] B. Bhargava and J. Riedl, "RAID Distributed Database System," *IEEE Trans. Software Eng.*, vol. 15, no. 6, pp. 726-736, 1989.

[11] B. Bhargava and J. Riedl, "A Formal Model for Adaptable Systems for Transaction Processing," *IEEE Trans. Knowledge and Data Eng.*, vol. 4, no. 1, pp. 433-449, 1989.

[12] S.B. Davidson, "Optimism and Consistency in Partitioned Distributed Database Systems," *Trans. Database Systems*, vol. 17, no. 3, pp. 456-481, ACM, Sept. 1984.

[13] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Relational Database System," *Comm. ACM*, vol. 8, no. 11, pp. 624-633, 1976.

[14] H. Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database," PhD thesis, Dept. of Computer Science, Stanford Univ., 1979.

[15] J.N. Gray, "Notes on Database Operating Systems" *Operating Systems: An Advanced Course*, R. Bayer, R.M. Graham, and G. Seegmuller, eds., Lecture Notes in Computer Science 60, Springer-Verlag, Heidelberg, Germany, 1978.

[16] J.N. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. VLDB Conf.*, Cannes, France, Sept. 1981.

[17] J.N. Gray and A. Reuter, *Transaction Processing: Concepts and Techniques*, Morgan Kaufmann, San Mateo, Calif., 1993.

[18] H.T. Kung and C.H. Papadimitriou, "An Optimality Theory of Database Concurrency Control," *Acta Informatica*, vol. 19, no. 1, pp. 1-13, 1984.

[19] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *Trans. Database Systems*, vol. 6, no. 2, pp. 213-226, ACM, 1981.

[20] L. Lamport, "Time Clocks and the Ordering of Events in a Distributed System," *Comm. ACM*, vol. 21, no. 2, 1979.

[21] P. Leu and B. Bhargava, "Multidimensional Timestamp Protocols for Concurrency Control," *IEEE Trans. Software Eng.*, vol. 13, no. 12, pp. 1,238-1,253, 1987.

[22] P. Leu and B. Bhargava, "Clarification of Two Phase Locking in Concurrent Transaction Processing," *IEEE Trans. Software Eng.*, vol. 14, no. 1, pp. 120-123, 1988.

[23] S.K. Madria and B. Bhargava, "System Defined Prewrites to Increase Concurrency in Databases," *Proc. First East European Symp. Advances in Databases and Information Systems*, St. Petersburg, Russia, ACM-SIGMOD, Sept. 1997.

[24] C.H. Papadimitriou, "The Serializability of Concurrent Database Updates," *J. ACM*, vol. 26, no. 4, pp. 631-653, 1979.

[25] C.H. Papadimitriou, *The Theory of Database Concurrency Control*, Computer Science Press, 1986.

[26] E. Pitoura and B. Bhargava, "Maintaining Consistency of Data in Mobile Distributed Environments" *Proc. 15th IEEE Int'l Conf. Distributed Computing Systems*, pp. 404-413, May 1995.+

[27] R. Prakash, M. Raynal, and M. Singhal, "An Adaptive Causal Ordering Algorithm Suited to Mobile Computing Environments," *J. Parallel and Distributed Computing*, pp. 190–204, Mar. 1997.

[28] A. Silberschatz and Z. Kedem, "Consistency in Hierarchical Database Systems," *J. ACM*, vol. 27, no. 1, pp. 72-80, 1979.

[29] R.H. Thomas, "A Majority Consensus Approach to Concurrency Control for Multiple Copy Systems," *Trans. Database Systems*, vol. 4, no. 2, pp. 180-209, ACM, 1979.

[30] J.D. Ullman, *Principles of Database Systems*, second ed., Computer Science Press, Potomac, Md., 1982.

[31] A. Zhang and A. Elmagarmid, "A Theory of Global Concurrency Control in Multidatabase Systems," *VLDB J.*, vol. 2, no. 3, pp. 331-359, July 1993.

[32] A. Zhang, M. Nordine, B. Bhargava, and O. Bukhres, "Ensuring Semi-Atomicity for Flexible Transactions in Multi-Database System," *Proc. SIGMOD Conf.*, pp. 67-78, Minneapolis, ACM, May 1994.

**Bharat Bhargava** graduated from the Indian Institute of Science and Purdue University in electrical and computer engineering. He is now a professor in the Department of Computer Science at Purdue. His research involves both theoretical and experimental studies in distributed systems. His research group has implemented a robust and adaptable distributed database system called RAID (for Reliable, Adaptable, Interoperable Distributed) to conduct experiments in replication control, checkpointing, and communications. He has conducted experiments in large-scale distributed systems, communications, and overheads in implementing object support on top of the relational model. He developed an adaptable video conferencing system using the NV system from Xerox PARC. He is currently conducting experiments with research issues in large-scale communication networks to support emerging applications, such as digital libraries and multimedia databases. He was chair of the IEEE Symposium on Reliable Distributed Systems, held at Purdue in October 1998. He is on the editorial board of three international journals. He and John Riedl received the Best Paper Award for their work, ”A Model for Adaptable Systems for Transaction Processing,” at the 1988 IEEE Data Engineering Conference. He received the Outstanding Instructor Award from the Purdue chapter of the ACM in 1996 and 1998. He is fellow of the IEEE and the Institute of Electronic and Telecommunication Engineering, and is a member of the ACM. He was named to the IEEE Computer Society Golden Core for distinguished service, and he has received the IEEE Computer Society's Meritorious Service Award.