Bangladesh Journal of Multidisciplinary Scientific Research; Vol. 1, No. 2
ISSN 2687-850X E-ISSN 2687-8518
July-September; 2019
Published by Centre for Research on Islamic Banking & Finance and Business

Concurrency Control in Database Systems

Dardina Tasmere
Senior Lecturer
Department of Computer Science and Engineering
Bangladesh Army University of Engineering & Technology, Natore, Bangladesh
E-mail: dardina.ruet@gmail.com

Md. Nazmus Salehin
B.Sc Student
Department of Computer Science and Engineering
Bangladesh Army University of Engineering & Technology, Natore, Bangladesh
E-mail: mdnazmussalehinnayan@gmail.com

Abstract
Concurrency control mechanisms including the wait, time-stamp and rollback mechanisms have been briefly discussed. The concepts of validation in optimistic approach are summarized in a detailed view. Various algorithms have been discussed regarding the degree of concurrency and classes of serializability. Practical questions relating arrival rate of transactions have been presented. Performance evaluation of concurrency control algorithms including degree of concurrency and system behavior have been briefly conceptualized. At last, ideas like multidimensional timestamps, relaxation of two-phase locking, system defined prewrites, flexible transactions and adaptability for increasing concurrency have been summarized.

Keywords: Concurrency, Control, Database Systems.

1. Introduction
Database systems are important for managing the data efficiently and allowing users to perform multiple tasks on it with the ease. From space station mechanism to credit card transactions, from railway station to telecommunications phonebook at our home – everywhere database is used. A database state which are the values of the database objects representing real-world entity is changed by the execution of a user transaction. In a distributed database system, the concurrency control problem occurs when several users access multiple databases on multiple sites. Correctness of the database is maintained by a scheduler that keeps an eye on the system and control the concurrent accesses. According to (Bhargava, 1983) database integrity and serializability measures the correctness of a database. A database is considered to be correct if a set of constraints (predicates or rules) are satisfied. Serializability ensures that a schedule for executing concurrent transactions is equivalent to one that executes the transactions serially in some order. It assumes that all accesses to the database are done using read and write operations. In this paper, we include some ideas that have been used for designing concurrency control algorithms and evaluated these algorithm's performance. Finally, we have taken into account some fact for increasing the degree of concurrency.

2. Concurrency Control Approaches and Algorithms
Our main point of focus is to process conflicting transactions correctly. Each transaction has a read and write set. Two transactions are said to be in conflictive they are different transactions; they are on the different set (one is read set and another is writing set) and/or they are on the same set where both of them are write sets. This concept is illustrated in Figure-1.

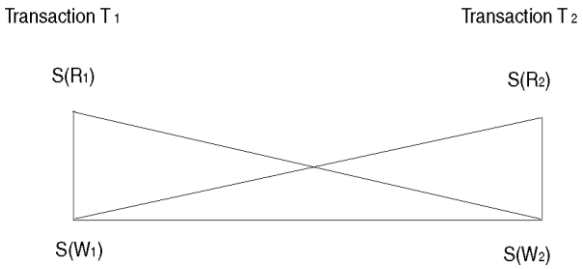


Figure 1. Types of conflicts for two transactions.

We say that the read set of T1 conflicts with the write set of T2 if read set S(R1) and write set S(W2) have some database entities (or items) in common. This shown by th diagonal edge in the figure. Similarly, if S(R2) and S(W1) have some database items in common, we draw the other diagonal edge. If S(W1) and S(W2) have some database items in common, we say that the write set of T1 conflicts with the write set of T2. It is shown by the horizontal edge at the bottom of the figure. As read actions do not change the values of the database entities, it is not a matter of concern about the confliction between the read sets of the two transactions. It needs to be pointed that if both transactions T1 and T2are executing at the same time, only then it can conflict. For example, if T2 was submitted to the system afterT1 has finished, they will not be in conflict even if their read and write sets intersect with each other.

Generic Approaches to Synchronization

In order to design concurrency control algorithms, basically there three generic approaches as follows:

- Wait: When two transactions conflict, one transaction must wait until another transaction is finished
- Timestamp: In this approach, a unique timestamp is assigned to every transaction by the system. Timestamp determines the order in which transactions will be executed. Conflicting actions of two transactions are processed in timestamp order. The time stamp can be assigned in the beginning, middle or at the end of the execution of a transaction.
- Rollback: If two transactions conflict with each other, some actions of one transaction are rolled back or it is restarted. This approach is also called optimistic because it is expected that conflicts are such that only a few transactions would roll back.

In the following section, we will discuss each of these approaches in detail and describe the concurrency control algorithms that are based on them.

2.1 Algorithms Based on Wait Mechanism

When two transactions conflict with each other, it can be solved by making one transaction wait until the other transaction releases the common entities. The system can use locking technique on the database entities in order to implement wait mechanism. The system can lock the entity as long as the operation continues and then can release the lock. When the lock has been given to some transaction, the requesting transaction must wait. Based on whether the transaction wants to do a read operation or a write operation, there are two types of locks that can be used to reduce the waiting time on an entity:

- **Read lock:** The transaction locks the entity in a shared mode. Any other transaction waiting to read the same entity can also obtain a read lock.
- **Write lock:** 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 read lock ora write lock.

When we say lock, it means either read lock or a write lock. When a transaction has completed operations on an entity, the transaction can perform 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. An important point is that lock and unlock operations can be performed by the user or be transparent to the transaction. In the latter case, it is the responsibility of the system to correctly perform locking and unlocking operations for each transaction.

Two new problems arise due to locking an entity. They Caerlaverock and deadlock. Live lock occurs when a transaction repeatedly fails to obtain a lock. Deadlock occurs when various transactions simultaneously tries to lock on several entities and as a result, 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 obtaining. The road to solution of the problem due to deadlock can be achieved by the following approaches

- Each transaction has to lock all entities at once. If some locks are held by some other transaction, then the transaction releases any locks that it was succeeded to secure.
- Assign an arbitrary linear ordering to the items, and all transactions need to request the locks based on this assigned order.

It has been observed that deadlocks in database systems are very rare and it may be cheaper to detect and resolve them rather than to avoid them (J.N. & Reuter, 1983).

As serializability is the correctness criterion for concurrently processing several transactions, locking must be done correctly to assure the above property. There is a protocol called Two-phase Locking (2PL). It is obeyed by all transactions in order to ensure serializability. This protocol says that in any transaction, all locks must precede all unlocks. A transaction operates in two phases: Locking phase is the first phase, and unlocking phase is the second phase. The first phase is also called the growing phase and the second phase is also called the shrinking phase. In the growing phase a transaction obtains more and more locks without releasing any locks. During the shrinking phase, the transaction releases more and more locks and is forbidden from obtaining additional locks. When the transaction terminates, all remaining locks are released automatically. The phenomenon just before releasing the first lock is called lock point. The Two-phase Locking and lock point are shown in Figure 2.

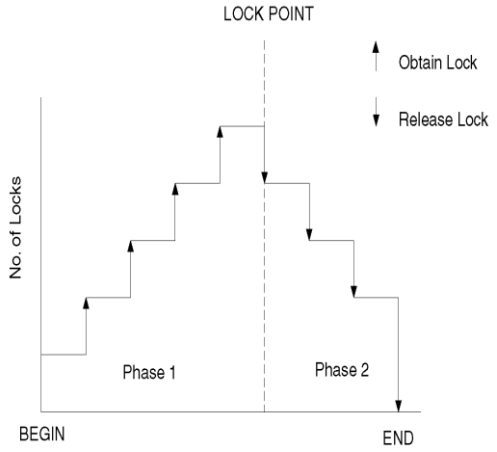


Figure 2. Two-phase locking and lock point: Obtain lock; — Release lock.

Now we discuss a centralized algorithm that utilizes locking in a distributed database system. We assume that all transactions write into the database and the database is fully replicated. A fully replicated database in real systems might not be very efficient. Besides, almost all the transactions usually read from the database only.

2.1.I A Sample Centralized Locking Concurrency Control Algorithm

Assume that a transaction Ti arrives at node X, then following steps are performed:

- Node X sends request to the central node for locking all the entities referenced by the transaction.
- The central node checks all the requested locks. Request is queued if some entity is already locked by another transaction. There is a queue for each entity. The request waits in one queue at a time.
- After receiving all its locks, transaction is executed at the central node. The values of read set are read from the database. Then required computations are carried out and the values of the write set are written in the database at the central node.
- If the database is fully replicated, the values of the write set are transmitted by the central node to all other nodes.
- Each node receives the new write set and updates the data base. Then central node receives an acknowledgment.
- When acknowledgment from all other nodes in the system is sent to the central nodes, it confirms that the transaction Ti has been completed at all nodes. Then the central node releases the locks and starts processing the next transaction.

There are some variations of the centralized locking algorithm. They are given bellow:

- Locking at Central Node, Execution at all Nodes: In this scenario, we assign the locks only at the central node and send the transaction back to the node X. Now transaction Ti 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 receives acknowledgments from all other nodes. Then it obtains the information that transaction Ti has been completed. The node X sends a message to unlock entities referenced by Ti. After receiving this message, the central node releases the locks and starts assigning locks to the waiting transactions which are queued.
- Avoid Acknowledgments, Assign Sequence Numbers: In the centralized control algorithm, the central node requires acknowledgments to ensure that the values of the write set have been written successfully at every node in the database. But the central node needs not to wait for this. If the central node guarantees that the write set values are written at every node in the same order as they were performed at the central node, the condition will be satisfied. 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. Besides equivalent efficiency is achieved.

Problems could occur for introducing sequence numbers. Let two transactions T5 and T6 are assigned sequence numbers 5 and 6 by the central node respectively. Assume that T5 and T6 have no common entities. That is why they do not conflict. For a long transaction T5, transactionT6, which arrived at the central node after T5, operation for T6 must wait at all nodes for T5 although T6 might be ready to write the values of its write set. This problem could be solved by attaching the sequence numbers of all lower-numbered transactions for which a given transaction have to wait before writing in the database. This list is called a wait-for list. 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. Sometimes the wait-for list size can grow very large, but transitivity among sequence numbers in wait-for lists can be used to reduce the size of it. Do-not-wait-for list, a complement of this wait-for list, can also be utilized. The details of wait-for list including many such ideas are discussed in (Garcia-Molina, 1979). Wait-for list is similar to causal ordering as discussed in (Prakash et al., 1997). In (Prakash et al., 1997) definition causal ordering is given as, If two messages M1 and M2 have the same destination and SEND(M1) ! SEND(M2), then DELIV ER(M1) ! DELIV ER(M2).

It is to be noted that there is a distinction between the reception of a message at a process and the delivery of the message to the corresponding process by the causal ordering protocol. Both message reception and delivery events are visible to the causal ordering protocol. However, the protocol hides the message reception event from the application process that uses the protocol. Thus, the application process is only aware of message delivery. According to the definition above, if M2 is received at the destination process before M1, the delivery of M2 to the process is delayed by the causal ordering protocol until after M1 has been received and delivered.

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 (Bhargava, B., & Hua, C. T.,1983).

- Global Two-phase Locking: This is a simple variation of the centralized locking mechanisms. In centralized locking, a transaction gets all locks in the beginning and release all locks in the end. In global two-phase locking, each transaction obtains the necessary locks on entities as they are needed. It releases locks 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.
- Primary Copy Locking: In this mechanism, 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.

Two-phase locking is a sufficient condition for serializability rather than the necessary condition. For example, if an entity is only used by a single transaction, it can be locked and unlocked independently. The question is, "How can we know this?" Since this information is not known to the individual transaction, so we are not aware of it. That is why, locking techniques are generally pessimistic.

2.2 Algorithms Based on Time-Stamp Mechanism

Timestamp is a mechanism where the serialization order is selected a priori and the transaction execution is bound to maintain this order. Each transaction is assigned to a unique timestamp by the concurrency controller. A timestamp on an entity represents the time when this entity was last updated.

All clocks at all nodes must be synchronized for obtaining unique timestamps at different nodes of a distributed system. Lamport (Lamport, 1979) 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. Then the local clock is incremented to the timestamp of the recently received message. In this way, all clocks become advanced until they are synchronized. There is another scenario where two identical timestamps must not be assigned to two transactions. In this case, at each tick of the clock, each node assigns a timestamp to only one transaction. In addition, the local clock time is stored in higher-order bits and the node identifiers are stored in the lower-order bits. Since node identifiers are different, this mechanism will ensure timestamps to be unique. When two transactions conflict with each other, they are needed to be processed in timestamp order. Thomas (Thomas, 1979) studied the correctness and implementation of this approach and described it. Timestamp order is used in each read-write conflict relation and write-write conflict relation. As all transactions have unique timestamps, it clarifies that cycles in a graph representing transaction histories are impossible.

2.2.1 Timestamp Ordering with Transaction Classes

Here, it is assumed that the read set and the write set of every transaction is known in advance. A transaction class is defined by a read set and a write set. 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 then it can be said that the transaction T is a member of a 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, mentioned in (Bernstein et al., 1980).

2.2.2 Distributed Voting Algorithm

The voting rule is the basis for concurrency control. Together with the request resolution rule it ensures that mutual exclusion is achieved for possibly conflicting concurrent updates. Two requests are said to conflict if the intersection of the base variables of one request and the update variables of the other request is not empty. This algorithm depends on distributed control to decide which transaction to be accepted and executed. Vote on each transaction is held in the nodes of the distributed database system. If a transaction gets a majority of OK votes, it is accepted for execution. When a transaction gets a majority of reject votes it is restarted. Nodes can also defer or postpone voting on a particular transaction. It is a result of the work of Thomas in (Thomas, 1979).

2.3 Algorithms Based on Rollback Mechanisms

In this section, a family of non-locking or optimistic concurrency control algorithms are discussed. Optimistic Methods for concurrency control are presented in detail in (Kunget al., 1981). Two families of non-locking concurrency controls are presented. The methods used are "optimistic" in the sense that they rely mainly on transaction backup as a control mechanism, "hoping" that

conflicts between transactions will not occur. Applications for which these methods should be more efficient than locking (Kung et al., 1981).

In this approach, the main concept is to validate a transaction against a set of previously committed transactions. In case the validation fails, the read set of the transaction is updated and computation is repeated 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 pre-analyze 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 considered to be pessimistic and decisions made at the end may invalidate the transaction processing and require rollback mechanism. There are four phases in the execution of a transaction in the optimistic concurrency control approach:

- Read: As reading an entity does not refer to a loss of integrity, reads are not restricted. A transaction reads the values of a set of entities 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 whereas the values of local variables are an indication of a past state of the database and are only known to the transaction. 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 $\Pi(Ri)$.
- Compute: The transaction computes 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.
- Validate: The transaction's local read set and write set are validated against a set of committed transactions. It is one of the main parts of this algorithm and is discussed in the next section.
- Commit and Write (called write for short): The transaction is said to be committed in the system if it succeeds in validation. Then it is assigned a timestamp denoted by $\Pi(Wi)$. On the other hand, 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.

2.3.1 The Validation Phase

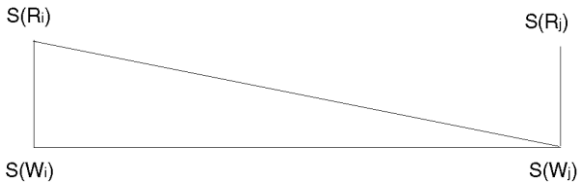
The concurrency controller can utilize syntactic information, semantic information, or a combination of the two. Here we discuss the use of syntactic information in the context of a validation at one node only. (Bhargava, B.,1983) discussed the use of semantic information.

Kung and Papadimitriou (Kung et al., 1984) 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.

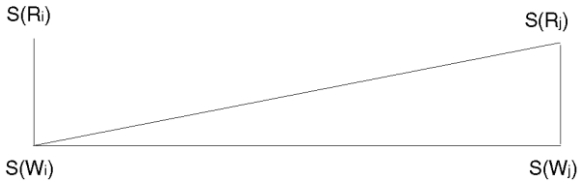
Consider two transactions Ti and Tj. Let $S(Ri)$ and $S(Ri)$ be the read sets and $S(Wi)$ and $S(Wj)$ be the write sets of Tiand Tj, respectively. Let $\Pi(Ri)$ and $\Pi(Rj)$ denote the time when the last item of the read set $S(Ri)$ and $S(Rj)$ were read from the database and let $\Pi(Wi)$ and $\Pi(Wj)$ denote the time when the first item of the write set $S(Wi)$ and $S(Wj)$ will be or were written in the database. Assume Ti arrives in the system before Tj. Let Tj be a committed transaction when the transaction Tj arrives for validation. Now there are four possibilities:

- If Tj and Tj do not conflict, Tj is successful in the validation phase and can either proceed or follow Tj.
- If $S(Ri) \cap S(Wi) \neq \emptyset$ and $S(Rj) \cap S(Wi) \neq \emptyset$, Ti fails in the validation phase and restarts.
- If $S(Ri) \cap S(Wj) \neq \emptyset$ and $S(Rj) \cap S(Wi) = \emptyset$, Tj is successful in validation. Tj must proceed Tj in any serial story since $\Pi(Ri) < \Pi(Wj)$. This possibility is illustrated as follows:



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

- If S(Ri) ∩ S(Wj) = Ø and S(Rj) ∩ S(Wi) ≠ Ø, Tj is successful in validation. Tj must follow Tj in any serial history since Π(Wi) > Π(Rj). 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 Ti and Tj do not conflict, do not draw any edge.
- If Ti must precede Tj, draw an edge from Tj to Tj,Tj→Tj.
- If Tj must follow Tj, draw an edge from Tj to Tj,Tj←Tj.

Thus, a directed graph is created for all committed transactions with transactions as nodes and edges as explained above. When a new transaction Tj arrives for validation, it is checked against each committed transaction to check if Tj 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 Ti be the validating transaction and let Tj 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 Tj that must be stored in memory must satisfy the following condition: If Tj is a committed transaction, store Tj for future validation if Ti∉ set of committed transaction such that:

$$\{\Pi(Ri) < \Pi(Wj)\} \text{ AND } \{S(Ri) \cap S(Wj) \neq \emptyset\}$$

It has been shown that the set of transactions to be stored for future validation will usually be small (Bhargava, B. K.,1982). 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 Tj can be deleted

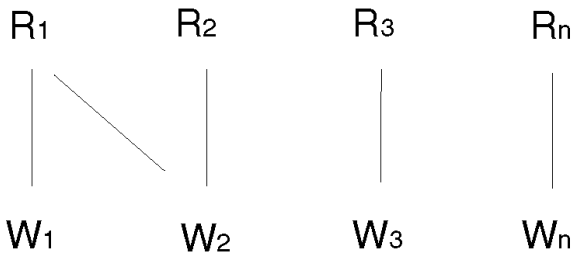
from this list. But care should be taken to restart (or invalidate) all active transactions Ti for which Π(Ri) < Π(Wj) before Tj is deleted.

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 Ri and Wi to represent the read action (as well as the read set) and the write action (as well as the write set) of a transaction Ti. Let h be an input history of n transactions to the scheduler as follows:

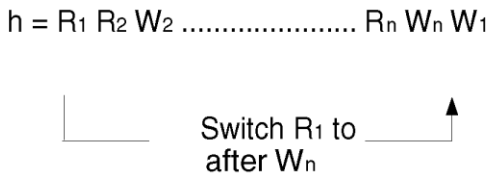
$$h = R1R2W2R3W3\ldots\ldots RnWnW1$$

Here transaction T1 executes the read actions, followed by the read/write action of T1, T2,….,Tn followed by the write actions of T1. Suppose R1 and W2 conflict as represented by an edge as follows:

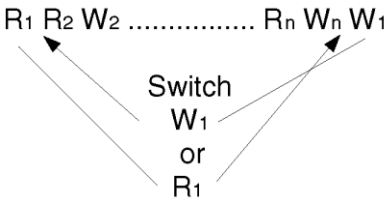


The history h is not allowed in the locking protocols because W2 is blocked by R1. If T1 is a long transaction and T2 is a small transaction, the response time for T2 will suffer. In general T1 can block T2, T2can block T3 (if T2 and T3 have a conflict) and so on. Let us consider several cases in optimistic approach.

Case 1: For the history h, in the optimistic approach of Kung and Robinson (Kung & Robinson, 1981) Ti(i = 2, …, n) can commit. Write sets (Wis) of committed transactions are saved to validate against the read set of T1. Basically the conflict preserving exchange (switch) as follows is attempted so that R1 can be brought next to W1.



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



The resulting histories can be either

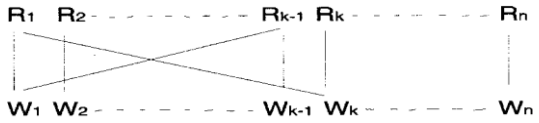
$$h = R1R2W2R3W3\ldots\ldots RnWnW1$$

Or

$$h = R2W2\ldots\ldots RnWnR1W1.$$

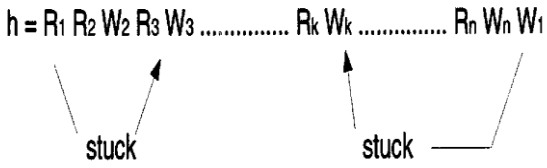
For switching W1, we would need to save not only the write sets of committed transactions, but also the read sets of committed transactions. This will allow more histories to be acceptable to the scheduler.

Case 3: A further extension of this idea is to try switching R1 toward W1 and W1 toward R1 if conflict preserving exchanges are possible. Consider the conflict as follows:

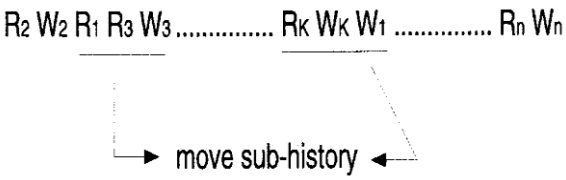


Consider the history h = R1R2W2…...Rk-1Wk-1Wk ……RnWnW1

Because of a conflict edge between R1 and Wk, R1can be scheduled only before WK. Similarly, due to the conflict edge Rk-1 and W1, W1 can be scheduled only after Rk-1. Switching R1 and W1, the scheduler can get a serializable history R2W2…. Rk-1Wk-1R1W1RkWk….RnWn. Using the switching of R1 or W1 alone would not have allowed this history to be acceptable to a scheduler. Finally, we consider the case where both R1 and W1 are stuck due to conflicts. Consider the history:



R1 can switch up to T3 and W1 can switch up to Tk due to conflicts (say R1W3 and W1Wk). The scheduler can try to move the sub-history R1R3W3 to the right and RkWkW1 to the left as shown next:



We can get a history as follows:

R2W2RkWk R1 W1 R3 W3….RnWn

which is serializable.

2.3.3 Implementation of Validations

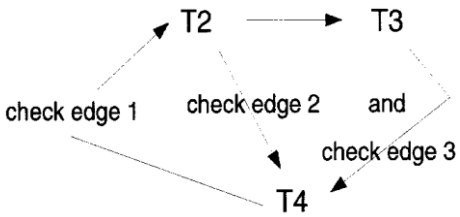
Let us now illustrate how these ideas for optimistic can be implemented. Consider n transactions. Assume T1 starts before T2, but finishes after Tn. Let T2T3 ..Tn finish in order. Since T2 is the first transaction to validate, it is committed automatically. So we have a conflict graph with a node for T2 as follows:

T2

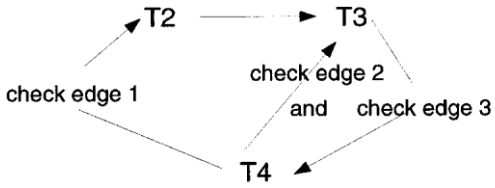
When T3 arrives for validation, the read set of T3 is validated against write set ofT2. If they conflict, the edge T3→T2 is drawn. Next, the write set of T3 is validated against the read set of T2 leading to the edge T2→T3. Since this causes a cycle, T3 is aborted. Otherwise, T3 is serialized with T2. So we have a conflict graph say as follows:

T2 – → T3

For a transaction T4 the edges are checked as follows: Check the edge T4→ T2. If it exists, check the edge T2→– T4. Abort if both edges exist. If only T4→ T2 exists, do not check the edge T4→ T3, but check the edge T3→T4only. This requires checking only three edges as follows:



If edge T4→ T2 does not exist, there is no need to check T2→ T4. In this case check the edge T4→ T3 and T3→ T4. 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 – I committed transaction. 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 shown by B. Bhargava in:(Bhargava,1984).

THEOREM I (Bhargava, 1984): 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.I 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 (Papadimitriou, 1979). The concurrency control algorithms for distributed database processing have been classified in (Bhargava et al., 1983). 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 (Bhargava & Hua, 1983). 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.

In contrast, the class DSS (distributed strict serializability) the histories retain the completion order of transactions based on the event w. 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 an acyclic conflict graph that is constructed for each history.

3.I.I The Hierarchy

All the classes G2PL, L2PL, DCP, DSTO, and DSS are serializable and form a hierarchy based on the degree of concurrency.

Figure. 3 depicts the hierarchy, where SR is the set of all serializable histories.

In Figure. 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:

Trans. 'a' = { Ra1[x]. Wa1[y].Wa2[y] };
Trans. 'b' = { Rb1[w]. Wb1[y]. Wb2[y] };
Trans. 'c' = { Rc2[u]. Wc1[v]. Wc2[v] };
Trans. 'd' = { Rd2[v]. Wd1[v]. Wd2[v] };
Trans. 'e' = { Re1[w]. We1[v]. We2[v] };
Trans. 'f' = { Rf2[t]. Wf1[w]. Wf2[w] };
Trans. 'g' = { Rg2[w]. Wg1[y]. Wg2[y] };

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 (Papadimitriou, 1979). 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: $R_e^1[w]R_f^2[t]W_f^1[w]W_f^2[w]R_d^2[v]W_d^2[v]W_d^1[v]$ $W_e^2[v]W_e^1[v]$
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

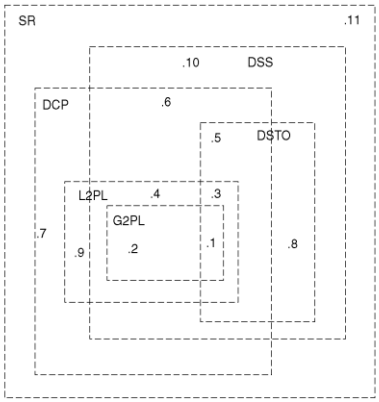


Figure 3. The hierarchy of the classes SR, DCP, L2PL, G2PL, DSTO, and DSS.

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 (Bhargava ,1982), 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 (Davidson, 1984), (Gray, 1981), (J.N. & Reuter, 1983). We present another strawman analysis. Assume that the database size is M and the read set and write set size is B. CBM 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:

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

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 (Bhargava, 1982) that illustrates the issues of arrival rate, relates multiprogramming level, frequency of cycles in a database environment. In Figure 4, we show that the degree of multiprogramming is low for a variety of transaction arrival rates in a sample database.

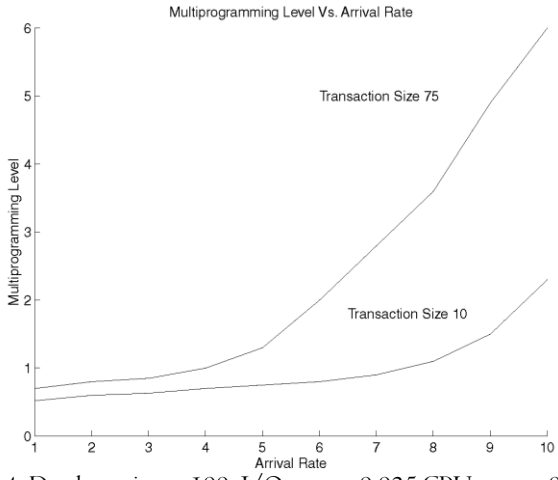


Figure 4. Database size = 100, I/O cost = 0.025,CPU cost = 0.0001.

In Figure 5, we show that the probability of a cycle is quite low for low degrees of multiprogramming. In Figure 6, we found that optimistic performs better than locking for very low arrival rates. Details of this study can be found in (Bhargava, 1982).

4. More Ideas for Increasing Concurrency

4.1 Multidimensional Time Stamps

There are several variations of timestamp ordering. For example, multiple versions (Papadimitriou, 1986) 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 (Leu & Bhargava, 1987) 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 foreach 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 (Papadimitriou, 1979).

4.2 Relaxations of Two-Phase Locking

In (Leu & Bhargava, 1988), 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 alater 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 transferal 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.

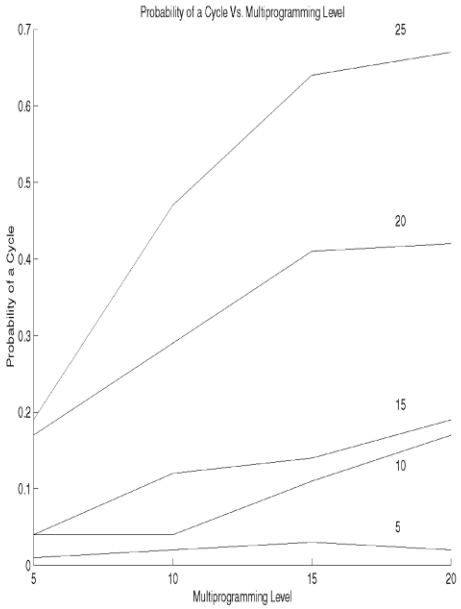


Figure 5. Database size = 500, transaction size = 5, 10, 15, 20, 25.

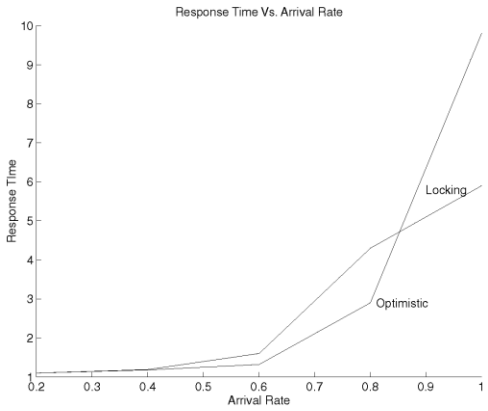


Figure 6. Database size = 200, no. of nodes = 3, communication delay = 0.10, I/O cost = 0.025, transaction size = 5 (time in seconds).

4.3 System Defined Prewrites

In (Madria & Bhargava, 1997) 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 aprecommit 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 non strict 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 (Zhang et al., 1994) supports flexible execution control flow by specifying two types of dependencies among the sub transactions of a global distributed transaction:

- Execution ordering dependencies between two sub transactions, and
- Alternative dependencies between two subsets of sub transactions.

A flexible transaction allows for the specification of multiple alternate subsets of sub transactions to be executed and results in the successful execution and commitment of the sub transactions in one of those alternate subsets, the execution of a flexible transaction can proceed in several different ways. The subtransaction in different alternate subsets maybe attempted simultaneously, as long as any attempted sub transactions not in the committed subset of sub transactions can either be aborted or have their effects undone. The flexible transaction model increases the failure resilience of global transactions. In (Zhang et al., 1994), we have defined a weaker form of atomicity, termed semi atomicity that is applicable to flexible transactions. Semi atomicity allows a flexible transaction to commit as long as a subset of its sub transactions that can represent the execution of the entire flexible transaction commit. Semi atomicity enlarges the class of executable global transactions in a heterogeneous distributed database system.

4.5 Adaptable Concurrency Control

Existing database systems can be interconnected that results 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 (Zhang & Elmagarmid, 1993). The issues of global serializability and deadlock resolution have been solved. The approach in (Bhargava & Riedl, 1989) 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 (Bhargava & Riedl, 1989) involves facilities to switch concurrency control methods. A formal model for an adaptable concurrency control (Bhargava & Riedl ,1989) 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. Conclusion

Concurrency Control problem occurs when several processes are concurrently involved in the system. Previously, serializability and two-phase locking were discussed in (Eswaran et al., 1976). (Eswaran et al., 1976) shows that, consistency requires that a transaction cannot request new locks after releasing a lock. Then it is argued that a transaction needs to lock a logical rather than a physical subset of the database. The ideas of timestamps were introduced by (Thomas, 1979).

The optimistic approach was proposed in (Kung & Robinson, 1981). Here, it is discussed that optimistic approaches rely mainly on transaction backup as a control mechanism, "hoping" that conflicts between transactions will not occur. In an optimistic approach, the major difficulty is starvation, which can be solved by using locking.

C.H. Papadimitriou has shown in (Papadimitriou, 1979) regarding the classes of serializability and the formalism for concurrency control. Several books that detail these subjects have been published (Bhargava, 1987), (Papadimitriou, 1986), (Bernstein et al., 1987), in addition to survey papers (Bernstein & Goodman, 1981), (Bhargava, 1983). The performance evaluation was studied in (Garcia-Molina, 1979). The ideas of adaptable concurrency control were published in (Bhargava & Riedl, 1989) and were implemented in the RAID system (Bhargava & Riedl, 1989). It has been commented by system experts that concurrency control only contributes5 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.

We continue to learn of new concepts including flexible transactions, value-dates, prewrites, degrees of commitment and view serializability discussed in (Bhargava, 1987). In large scale systems, it is difficult to block access for transactions using locking mechanism to database objects. We encourage readers to learn from various books on both theory and implementation of concurrency control mechanisms.

References

Bernstein, P. A., & Goodman, N. (1981). Concurrency control in distributed database systems. *ACM Computing Surveys (CSUR)*, 13(2), 185-221.

Bernstein, P. A., Hadzilacos, V., & Goodman, N. (1987). Concurrency control and recovery in database systems.

Bernstein, P. A., Shipman, D. W., & Rothnie Jr, J. B. (1980). Concurrency control in a system for distributed databases (SDD-1). *ACM Transactions on Database Systems (TODS)*, *5*(1), 18-51.

Bhargava, B. K. (1982, October). Performance Evaluation of the Optimistic Approach to Distributed Database Systems and Its Comparison to Locking. In *ICDCS* (pp. 508-517).

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

Bhargava, B. (1987). Concurrency Control and Reliability in Distributed Systems," B. Bhargava, ed., Van Nostrand and Reinhold, 1987.

Bhargava, B., & Hua, C. T. (1983). A causal model for analyzing distributed concurrency control algorithms. *IEEE transactions on software engineering*, (4), 470-486.

Bhargava, B. (1984). Resilient concurrency control in distributed database systems. *IEEE transactions on reliability*, *31*(5), 437-443.

Bhargava, B. (1987). Transaction processing and consistency control of replicated copies during failures in distributed databases. *Journal of Management Information Systems*, *4*(2), 93-112.

Bhargava, B., & Riedl, J. (1989). The Raid distributed database system. *IEEE Transactions on Software Engineering*, *15*(6), 726-736.

Bhargava, B., & Riedl, J. (1989, February). A Formal model for adaptable systems for transaction processing. IEEE Trans. Knowledge and Data Eng.*4*(1), 433-449.

Davidson, S. B. (1984). Optimism and consistency in partitioned distributed database systems. *ACM Transactions on Database Systems (TODS)*, *17*(3), 456-481.

Eswaran, K. P., Gray, J. N., Lorie, R. A., & Traiger, I. L. (1976). The notions of consistency and predicate locks in a database system. *Communications of the ACM*, *8*(11), 624-633.

Garcia-Molina, H. (1979). *Performance of Update Algorithms for Replicated Data in a Distributed Database* (No. STAN-CS-79-744). STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE.

Gray, J. N. (1978). Notes on database operating systems. *Operating Systems: An Advanced Course*, *60*, 397-405.

Gray, J. (1981, September). The transaction concept: Virtues and limitations. In *VLDB* (Vol. 81, pp. 144-154).

J.N. Gray & Reuter, A. (1983). Transaction Processing: Concepts and Techniques, Morgan Kaufmann, San Mateo, Calif.

Kung, H. T., & Papadimitriou, C. H. (1984). An optimality theory of database concurrency control. *Acta Informatica, 19*(1), 1-13.

Kung, H. T., & Robinson, J. T. (1981). On optimistic methods for concurrency control. *ACM Transactions on Database Systems (TODS), 6*(2), 213-226.

Lamport, L. (1979). Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM, 21*(7), 558-565.

Leu, P. J., & Bhargava, B. (1987). Multidimensional timestamp protocols for concurrency control. *IEEE Transactions on Software Engineering*, 13(12), 1238-1253.

Leu, P. J., & Bhargava, B. (1988). Clarification of two phase locking in concurrent transaction processing. *IEEE transactions on software engineering*, *14*(1), 120-123.

Madria, S. K., & Bhargava, B. K. (1997, September). System Defined Prewrites for Increasing Concurrency in Databases. In *ADBIS* (pp. 18-22).

Papadimitriou, C. H. (1979). *Serializability of concurrent database updates* (No. MIT/LCS/TR-210). MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTER SCIENCE.

Papadimitriou, C. (1986). The theory of database concurrency control. Computer Science Press

Pitoura, E., & Bhargava, B. (1995, May). Maintaining consistency of data in mobile distributed environments. In *Proceedings of 15th International Conference on Distributed Computing Systems* (pp. 404-413). IEEE.

Prakash, R., Raynal, M., & Singhal, M. (1997). An adaptive causal ordering algorithm suited to mobile computing environments. *Journal of Parallel and Distributed Computing*, *41*(2), 190-204.

Silberschatz, A., & Kedem, Z. (1979). Consistency in hierarchical database systems. *Journal of the ACM (JACM), 27*(1), 72-80.

Thomas, R. H. (1979). *A majority consensus approach to concurrency control for multiple copy Systems.* Trans. Database Systems, ACM, *4*(2), 180-209

Ullman, J. D. (1982). *Principles of database systems*. second ed., Computer Science Press, Potomac, Md.

Zhang, A., & Elmagarmid, A. K. (1993). A theory of global concurrency control in multidatabase systems. *The VLDB Journal—The International Journal on Very Large Data Bases*, *2*(3), 331-359.

Zhang, A., Nodine, M., Bhargava, B., & Bukhres, O. (1994). *Ensuring Semi-Atomicity for Flexible Transactions in Multi-Database System* (Vol. 23, No. 2, pp. 67-78). ACM.