A general concurrency control for database systems

by ABDEL A. FARRAG and M. TAMER OZSU
The University of Alberta
Edmonton, Alberta

ABSTRACT

The concurrency control problem in database systems has been examined by many people and several concurrency control algorithms have been proposed. The most popular concurrency controls are two-phase locking and timestamp ordering. This paper reviews both algorithms and shows that they are special cases of a more general concurrency control algorithm. This concurrency control algorithm is described in detail and is proven to work correctly. Several other special cases of the proposed mechanism are presented. We show that two-phase locking and timestamp ordering represent the two end points of a series of concurrency controls. Each of them is a special case of the general concurrency control described in this paper.
INTRODUCTION

The problem of coordinating concurrent accesses to a database system has been studied by many people and several concurrency control algorithms have been introduced. The task of a concurrency control is to ensure the consistency of the database while allowing a set of transactions to execute concurrently. A database is a collection of entities named \( x, y, z, \) etc. Each entity has a single value. The consistency of the database is defined in terms of a set of constraints called integrity constraints. The database is said to be consistent if the values of its entities satisfy the predefined set of integrity constraints. A transaction is any sequence of read and write operations which preserves the consistency of the database. Each action is either a read or write operation on some entity of the database. In order to allow a set of transactions to execute concurrently, a concurrency control is needed to resolve the conflicts among transactions and to ensure that the overall effect of their execution is correct (i.e., transforms a consistent database state into a new consistent state).

Serializability still remains the main correctness criterion for concurrency control. A concurrent execution of a set of transactions (or schedule) is said to be serializable if it is equivalent to (i.e., has the same effect on the database as) a serial schedule in which the transactions are executed sequentially in some order. The concurrency control performs its job by producing only serializable schedules. The most popular (and practical) concurrency controls are two-phase locking and timestamp ordering. This paper reviews both mechanisms and shows that they are special cases of a more general concurrency control algorithm. This concurrency control algorithm is described in detail and is proven to work correctly. Several other special cases of the proposed mechanism are also presented. The set of transactions executing at any time under the proposed concurrency control forms a set of classes called strict classes. The maximum number of transactions which can belong to the same strict class is called the strictness level of the concurrency control. The value of the strictness level can be specified in advance and can be modified during execution. The higher the value of the strictness level the more strict the concurrency control and the lower the value of the strictness level the less strict the concurrency control.

TWO-PHASE LOCKING AND TIMESTAMP ORDERING

This section defines some basic concepts which will be used throughout this paper and reviews two-phase locking and timestamp ordering mechanisms.

Definition 1 (Transaction): A transaction is any sequence of read and/or write operations which preserves the integrity constraints of the database. A read operation by a transaction \( T \) on an entity \( x \) is denoted \( R_T(x) \). Similarly, a write operation on \( x \) by \( T \), is denoted \( W_T(x) \). A transaction is assumed to be a correct computation (i.e., either all of its updates must be reflected in the database or none of them). A transaction is said to be terminated or committed when all of its operations have been accepted by the concurrency control. If an operation by the transaction is rejected, the transaction must be aborted.

Definition 2 (Schedule): A schedule \( S \) of a set of transactions \( T = \{ T_1, T_2, \ldots, T_n \} \) is an interleaved sequence of the operations of the transactions in \( T \). When the operations of each transaction appear consecutively (i.e., without interleaving with the operations of the other transactions), the schedule is said to be serial.

Definition 3 (Conflicting operations): Two operations by two different transactions are said to be in conflict if both operations access the same entity and one of them is a write operation. This definition implies that read operations do not conflict.

Definition 4 (Dependency graph): The dependency graph \( DG(S) \) of a schedule \( S \) of a set of transactions \( T = \{ T_1, T_2, \ldots, T_n \} \) is a directed graph which represents the conflict relations among transactions. The set of nodes of this graph is \( \{ T_1, T_2, \ldots, T_n \} \), and an arc form \( T_i \) to \( T_j \) exists in the graph iff there is an operation by \( T_i \) which conflicts with and precedes (i.e., its order in the schedule comes before) another operation by \( T_j \).

Definition 5 (Equivalence of two schedules): Two schedules \( S_1 \) and \( S_2 \) for the same set of transactions are said to be equivalent if \( DG(S_1) = DG(S_2) \). A schedule \( S \) is said to be serializable iff it is equivalent to a serial schedule.

Theorem 1

If the dependency graph \( DG(S) \) of a schedule \( S \) is acyclic, then \( S \) is serializable.

Two-Phase Locking

Locking is the most commonly used mechanism for controlling concurrency. Locking requires each transaction before reading or writing an entity \( x \) to obtain a read-lock or a write-lock on \( x \), respectively. Several transactions can obtain a read-lock on \( x \) simultaneously, but no more than one transaction can obtain a write-lock on \( x \) at the same time. When a read- or write-lock on some entity cannot be granted because the
entity is already locked, the transaction which requested the lock has to wait until the lock is released.

The most famous locking protocol is called two-phase locking. Two-phase locking guarantees serializability by preventing a transaction from obtaining a lock on any entity after releasing a lock on any other entity. Therefore, each transaction has two phases, one during which locks can only be obtained followed by another one during which locks can only be released. The point at which the transaction releases its first lock delimits the two phases. This point is called the locked point of the transaction.

Some systems do not require a transaction to use explicit commands (like Lock(x) and Unlock(x)) to lock and release the entities it accesses. Instead, a read-lock or a write-lock is requested implicitly when the transaction submits a read or write operation, respectively, to the concurrency control. This lock will be granted if the operation is accepted; otherwise the operation is placed on a waiting queue. When all the operations of the transaction succeed (i.e., have been accepted), all locks are released by a single atomic action. This form of two-phase locking is called strict two-phase locking because each transaction maintains all locks until termination. Throughout this paper, the term two-phase locking will refer to a strict two-phase locking mechanism in which read- and write-locks are granted and released as described above.

Two-phase locking has some drawbacks. Because transactions can be arbitrarily long (but finite) and each transaction must maintain all the locks it obtains during execution until its very end, two-phase locking reduces the level of concurrency. Another drawback is the problem of deadlock. Deadlock can occur because transactions wait for one another. In order to detect deadlock, a directed graph called the wait for graph is maintained. The nodes of the graph represent the transactions and the arcs represent the wait-for relationship (i.e., an arc Ti→Tj exists if Ti is waiting for Tj to release its locks on some entity. A deadlock occurs when the graph contains a cycle. In order to resolve the problem one or more transaction must be aborted.

Theorem 2

Two-phase locking is a correct synchronization mechanism (i.e., every schedule produced by a two-phase locking mechanism is serializable).

Timestamp Ordering

In a timestamp ordering mechanism, each transaction is assigned a unique timestamp (i.e., number) when it starts. The timestamp of Ti will be attached to every read or write operation issued by Ti. For each entity x, two values are recorded, TSR(x) and TSW(x). These values record the largest timestamp of any read and write operations processed on x, respectively. Timestamp ordering guarantees serializability for executing all conflicting operations in timestamp order.

The basic timestamp ordering mechanism processes read and write operations as follows. When a read request R(x) by a transaction Ti is received, the timestamp of Ti is compared with the value of TSW(x). If it is smaller, the operation is rejected. Otherwise, the operation is accepted. Similarly, when a write operation W(x) is received, its timestamp is compared with the value max(TSR(x), TSW(x)). If it is smaller, the operation is rejected. Otherwise, it is accepted.

When an operation is rejected, the transaction which issued the operation must be aborted. Aborting a transaction involves undoing all steps which have been executed and restarting the transaction from the beginning. Abortion is the main drawback of the basic timestamp ordering. If it occurs frequently, performance can be degraded. Other timestamp ordering mechanisms which necessitate lesser abortion than the basic timestamp ordering mechanism described above have been proposed, but in this paper, we are only concerned with the basic timestamp ordering.

The basic timestamp ordering mechanism has the advantage of being deadlock free. It also achieves a higher level of concurrency than two-phase locking because transactions never "wait."

Theorem 3

Timestamp ordering is a correct synchronization mechanism.

THE PROPOSED CONCURRENCY CONTROL

This section describes the proposed concurrency control and proves it works correctly (i.e., produces only serializable schedules).

Transaction Timestamps

Each transaction Ti is assigned a unique timestamp t1(Ti), when it starts. The timestamp t1(Ti) is a pair (t1(Ti), t1(Ti)), where t1(Ti) is called the global timestamp of Ti and t1(Ti) is called the local timestamp of Ti. Timestamps are assigned using the method described later. The method assumes that there is some positive integer L representing the maximum number of transactions which can have the same global timestamp simultaneously. We will call L the strictness level of the concurrency control for a technical reason which will become apparent later. For the purpose of this section, we assume that the value of L is fixed (i.e., does not change during execution). How the value of L is chosen and its impact on the performance of the concurrency control will be discussed in the next section. In this section, we assume an arbitrary value for L.

The method used for assigning timestamps uses four counters, C1, C2, C3, and C4. The current value of each counter C1 is denoted by V(C1). C1 and C2 are used to generate the global and local timestamps, respectively. C3 is used to count the number of transactions currently executing and having the global timestamp equal to V(C1). C4 is used to keep track of the current number of executing transactions. V(C4) must always be ≤ M where M is some integer representing the multiprogramming level of the system (i.e., the maximum
number of transactions which can be executed concurrently. It is assumed that if the number of executing transactions equals \( M \), then no other transaction can start execution until one of the currently executing transactions is terminated or aborted. Assigning timestamps proceeds as described below. (Initially, all counters contain the value 0).

Starting or restarting a transaction

When a new transaction \( T_i \) arrives in the system (or when \( T_i \) is restarted) the values of \( t_f(T_i) \) and \( t_d(T_i) \) are assigned as follows:

1. If \( V(C_i) = M \), then \( T_i \) cannot be started (i.e., put \( T_i \) on a waiting queue and stop). Otherwise, \( V(C_i) = V(C_i) + 1 \).
2. If \( V(C_i) < L \), then \( V(C_i) = V(C_i) + 1 \) and \( V(C_i) = V(C_i) + 1 \). Otherwise, \( V(C_i) = V(C_i) + 1 \).
3. \( t_f(T_i) = V(C_i) \) and \( t_d(T_i) = V(C_i) \).

Terminating or aborting a transaction

When a transaction \( T_i \) is terminated or aborted, the values of \( V(C_i) \) and \( V(C_i) \) will be modified as follows:

1. If \( t_f(T_i) = V(C_i) \) at the time of termination or abortion, then \( V(C_i) = V(C_i) - 1 \).
2. \( V(C_i) = V(C_i) - 1 \).

For each entity \( x \), the concurrency control maintains the following values:

i) \( \text{GTSW}(x) \) and \( \text{LTSW}(x) \)

\( \text{GTSW}(x) \) is a variable which records the largest global timestamp of any transaction that wrote \( x \). This value is recorded when the write operation of the transaction is accepted. The local timestamp of such a transaction is recorded at the same time in the list \( \text{LTSW}(x) \). If the transaction is terminated (or aborted) and its global and local timestamps are still recorded in \( \text{GTSW}(x) \) and \( \text{LTSW}(x) \), respectively, the local timestamp is deleted from \( \text{LTSW}(x) \).

ii) \( \text{GTSR}(x) \) and \( \text{LTSR}(x) \)

\( \text{GTSR}(x) \) is a variable which records the largest global timestamp of any transaction that read \( x \). This value is recorded when the read operation of the transaction is accepted. The local timestamp of the transaction is recorded at the same time in the list \( \text{LTSR}(x) \). In general, several transactions with the same global timestamp (or with different global timestamps) can read \( x \) simultaneously. \( \text{LTSR}(x) \) records the local timestamp of every transaction that read \( x \) and whose global timestamp is recorded in \( \text{GTSR}(x) \). If a transaction is terminated (or aborted) and its global and local timestamps are recorded in \( \text{GTSR}(x) \) and \( \text{LTSR}(x) \), respectively, the local timestamp is deleted from \( \text{LTSR}(x) \). (Note that \( \text{LTSR}(x) \) may contain several values, each representing the local timestamp of some transaction.) Initially, i.e., before starting execution, \( \text{LTSW}(x) \) and \( \text{LTSR}(x) \) are empty, and \( \text{GTSW}(x) \) and \( \text{GTSR}(x) \) record the initial value of the counter \( C_i \).

The following subsections describe how the concurrency control processes read and write operations and prove that the concurrency control produces only serializable schedules.

Processing Read Operations

When the concurrency control receives a read operation \( R_i(x) \), one of the following cases arises:

1. \( t_f(T_i) < \text{GTSW}(x) \). This means that \( x \) has been written by a transaction which has a larger global timestamp than \( T_i \). In this case, \( R_i(x) \) is rejected.
2. \( t_f(T_i) > \text{GTSW}(x) \). This means that \( x \) has not been written by any transaction which has a larger global timestamp than \( T_i \). In this case, \( R_i(x) \) is accepted.
3. \( t_f(T_i) = \text{GTSW}(x) \). This means that \( x \) has been written by a transaction which has the same global timestamp as \( T_i \). When this case occurs, one of the following conditions is true:
   a. \( \text{LTSW}(x) \) is empty. This means that any transaction that wrote \( x \) and has the same global timestamp as \( T_i \) has been terminated. In this case \( R_i(x) \) is accepted.
   b. \( \text{LTSW}(x) \) is not empty. This means that \( x \) has been written by another transaction which has the same global timestamp as \( T_i \) and that this transaction has not been terminated. In this case \( R_i(x) \) has to wait.

In general, when a read or write operation has to wait because it conflicts with a previously granted operation by a different transaction having the same global timestamp, the operation cannot be processed until the transaction is terminated or aborted (and, therefore, the conflict no longer exists). If an operation with a larger global timestamp than the waiting operation and in conflict with it has been accepted, the waiting operation will be rejected (i.e., deleted from the waiting queue). The wait-for relationship among transactions can be represented by a directed graph similar to the one described previously.

Processing Write Operations

When the concurrency control receives a write operation \( W_i(x) \), one of the following cases arises:

1. \( t_f(T_i) < \max(\text{GTSR}(x), \text{GTSW}(x)) \). This means that \( x \) has been read or written by any other transaction which has a larger global timestamp than \( T_i \). In this case, \( W_i(x) \) is rejected.
2. \( t_f(T_i) > \max(\text{GTSR}(x), \text{GTSW}(x)) \). This means that \( x \) has not been read or written by any other transaction which has a larger global timestamp than \( T_i \). In this case, \( W_i(x) \) is accepted.
3. \( t_f(T_i) = \max(\text{GTSR}(x), \text{GTSW}(x)) \). This means that \( x \) has been read or written by one or more transactions
having the same global timestamp as \( T_i \). When this case occurs, one of the following conditions is true:

a. \( \text{GTSW}(x) > \text{GTSR}(x) \). In this case, the concurrency control examines \( \text{LTSW}(x) \). If \( \text{LTSW}(x) \) is empty, then \( W_i(x) \) is accepted; otherwise, \( W_i(x) \) has to wait.

b. \( \text{GTSW}(x) < \text{GTSR}(x) \). In this case, the concurrency control examines \( \text{LTSR}(x) \). If \( \text{LTSR}(x) \) is empty or contains only the local timestamp of \( T_i \), then \( W_i(x) \) is accepted; otherwise, \( W_i(x) \) has to wait.

c. \( \text{GTSW}(x) = \text{GTSR}(x) \). In this case, the concurrency control examines \( \text{LTSW}(x) \) and \( \text{LTSR}(x) \). If \( \text{LTSW}(x) \) is empty and \( \text{LTSR}(x) \) is empty or contains only the local timestamp of \( T_i \), then \( W_i(x) \) is accepted; otherwise, \( W_i(x) \) has to wait.

\[ \text{Theorem 4} \]

Every schedule produced by the concurrency control is serializable.

\[ \text{Proof:} \] Let \( S \) be a schedule produced by the concurrency control. An arc \( T_a \rightarrow T_b \) in \( \text{DG}(S) \) indicates that one of the following two conditions is true: either \( t_p(T_a) < t_p(T_b) \) or \( t_p(T_b) < t_p(T_a) \) and \( T_b \) terminates before \( T_a \). Let \( T_i \) and \( T_j \) be two arbitrary nodes such that there is a path (of length greater than zero) from \( T_i \) to \( T_j \) in \( \text{DG}(S) \). Then, by transitivity, either \( t_p(T_i) < t_p(T_j) \) or \( t_p(T_j) < t_p(T_i) \) and \( T_i \) terminates before \( T_j \). Let \( T_k \) be another path from \( T_i \) to \( T_j \) and let \( T_k \) terminate before \( T_i \). In either case, another path from \( T_i \) to \( T_j \) cannot exist because it leads to a contradiction. Therefore, \( \text{DG}(S) \) is acyclic (i.e., \( S \) is serializable).

\[ \text{THE STRICTNESS LEVEL} \]

The set of transactions executing at any time under the concurrency control can be partitioned into a set of disjoint classes; each class containing the set of transactions that have the same global timestamp. Let us call these classes strict classes. The set of strict classes changes dynamically during execution (i.e., when a transaction is started, terminated or aborted). In the previous section, we assumed that the maximum number of transactions that can have the same global timestamp simultaneously during execution (i.e., can belong to the same strict class) is chosen arbitrarily. We called this number the strictness level of the concurrency control. This section describes the impact of the strictness level on the performance of the concurrency control.

Consider a special case of the concurrency control described in the previous section in which the value of the strictness level \( L \geq M \), where \( M \) is the multiprogramming level of the system (defined in the previous section). In this case, the set of executing transactions will always have the same global timestamp (i.e., the number of strict classes at any time during execution will equal 1) because the number of executing transactions cannot exceed the value of \( M \). It is not difficult to see that in this particular case processing read and write operations is performed by the concurrency control as in the two-phase locking mechanism described earlier. As in the procedures described in the previous section for processing read and write operations, the conditions for rejecting an operation will never be satisfied. In this case, the concurrency control will respond to each read or write operation by either accepting or delaying the operation depending on the condition satisfied when the operation is received.

Consider another special case in which \( L = 1 \). In this case, the set of executing transactions will always have different global timestamps (i.e., the number of strict classes at any time during execution will equal the number of transactions executing at that time). It is not difficult to see that in this particular case processing read and write operations is performed by the concurrency control as in the timestamp ordering mechanism described earlier. As in the procedures described previously for processing read and write operations, the conditions for delaying an operation will never be satisfied. In this case, the concurrency control will respond to each read or write operation by either accepting or rejecting the operation depending on the condition satisfied when the operation is received. This proves the following lemma.

\[ \text{Lemma 1} \]

Two-phase locking and timestamp ordering are special cases of the concurrency control mechanism proposed in this paper.

Several other special cases of the proposed mechanism arise for \( L = 2, 3, \ldots, M - 1 \). Processing read and write operations in each of these cases is performed by the concurrency control as if it were a combination of both mechanisms, two-phase locking and timestamp ordering. In this case, an operation will be processed relative to another operation having the same global timestamp as if the concurrency control is two-phase locking, and relative to another operation having a different global timestamp as if the concurrency control is timestamp ordering. The following lemma gives the minimum and the maximum number of strict classes at any time during execution for any value of \( L \), where \( 1 \leq L \leq M \).

\[ \text{Lemma 2} \]

Let \( L, M, V(C_4) \) and \( C \) refer to the strictness level, the multiprogramming level, the number of executing transactions and the number of strict classes at any given time during execution, respectively. Then, \( [V(C_i)/L] \leq C \leq \min(V(C_i), M - L + 1) \).

\[ \text{Proof:} \] The proof of the above lemma is an immediate consequence of the method described previously for assigning timestamps. The minimum number of strict classes corresponds to a situation in which no more than one class has fewer than \( L \) transactions. The maximum number of strict classes corresponds to a situation in which \( M - L \) transactions (or every transaction if \( V(C_i) \leq L \)) will belong to \( M - L \) different classes and the remaining transactions will belong to another strict class. (The reader must convince himself that the above situations can occur.)

Therefore, two-phase locking and timestamp ordering represent the two end points of a series of concurrency controls.
Each of them is a special case of the general concurrency control mechanism described in this paper (i.e., for every different value of $L$, where $1 \leq L \leq M$, a different special case results). Each of these special cases has a different level of concurrency. In particular, two-phase locking and timestamp ordering have the lowest and highest level of concurrency, respectively. The value of the strictness level is, in some sense, a measure of the level of concurrency. The greater the value of $L$, the stricter the concurrency control. Although we assumed in the previous section that the value of the strictness level is fixed during execution, this is not necessary in fact. The value can change from time to time during execution (without affecting the correctness of the concurrency control).

Some researchers have already shown that if the level of transaction conflict is low, other concurrency control mechanisms which allow a higher level of concurrency will perform better than locking. We believe that the performance of a concurrency control mechanism must depend on both the likelihood of transaction conflict and the level of concurrency allowed by the mechanism. The exact relationship between performance and transaction conflict and concurrency has not been formalized and deserves more research.

This paper has proposed a general concurrency control mechanism in which the strictness level (or, in some sense, the concurrency level) can be specified and even modified during execution. The performance issue, which has not been examined in detail, is the subject of further research. A problem which has not been discussed is that of deadlock. Deadlock can only occur if the strictness level is greater than 1. Moreover, it only involves transactions which have the same global timestamp because a transaction never waits for any other transaction which has different global timestamp. Deadlock can be detected by maintaining a wait-for graph similar to the one described previously.

CONCLUSIONS

Two-phase locking and timestamp ordering are the most popular and practical concurrency controls. Both mechanisms have been reviewed and shown to be special cases of a more general concurrency control algorithm. This concurrency control algorithm has been described and proven to work correctly (i.e., produces only serializable schedules). Two-phase locking and timestamp ordering represent the two end points of a series of concurrency controls. Each results from choosing a different value for the strictness level and is considered to be a special case of the general concurrency control described in this paper.

REFERENCES
