Chapter 2
Concurrency Control Methods

2.0 Introduction

In this chapter we consider concurrency control methods in monoversion distributed database systems. These methods are the most widely found in existing DDBSs. In section 2.1 we present the methods based on locking, timestamp ordering, and optimistic approach. In section 2.2 Two-Phase commit protocol that will be used when the transaction commits is described.

2.1 Concurrency Control Methods and Algorithms

There are many concurrency control algorithms for distributed database in the literature. Most of these distributed concurrency control algorithms fall into one of three basic classes: Locking algorithms [Bern81, Gray78, Mena78, Rose78, Ston79, Trai82], Timestamp algorithms [Bern80a, Reed83, Thom79], and optimistic algorithms [Bada79, Ceri82, Kung81, Sch181, Sinh85]. Bernstein and Goodman [Bern81] review many of the proposed algorithms and describe how additional algorithms may be synthesized by combining basic mechanisms from the locking and timestamp classes.

In the succeeding paragraphs the most widely used concurrency control methods in DDBMS and some of the concurrency control algorithms based on these methods are reviewed.

2.1.1 Locking Method

2.1.1.1 Locking Concept

Locking is a mechanism commonly used to solve the problem of synchronizing access to shared data. The idea behind locking is intuitively simple. Each data item has
a lock associated with it. Before a transaction $T_1$ may access a data item, the scheduler first examines the associated Lock. If no transaction holds the lock, then the scheduler obtains the lock on behalf of $T_1$. If another transaction $T_2$ does hold the lock, then $T_1$ has to wait until $T_2$ gives up the lock. That is, the scheduler will not give $T_1$ the lock until $T_2$ releases it. The scheduler thereby ensures that only one transaction can hold the lock at a time, so only one transaction can access the data item at a time [Bern87].

Typical locking mechanisms are complex, since they have the notion of lock mode: a transaction locks a data item in shared mode, if it wants to read the data item, and in exclusive mode, if it wants to write the data item. A transaction is well-formed, if it always locks a data item in shared mode before reading it, and it always locks a data item in exclusive mode before writing it. The correctness of locking mechanisms is based on the assumption that transactions are well-formed [Bern79].

The following compatibility rules exist between locks modes [Papd79]:

1. A transaction can lock a data item in shared mode if it is not locked at all or it is locked in shared mode by another transaction.
2. A transaction can lock a data item in exclusive mode only if it is not locked at all.

Two transactions are in conflict, if (a) they are on the same data item (b) they are issued by different transactions (c) one or both of them are in exclusive mode.

### 2.1.1.2 Two-Phase Locking Algorithm

In this section we present the most used algorithm of the locking method called two-phase locking (2PL) [Bern81, Eswa76]. The idea of this algorithm is the following: the last data item has to be locked by a transaction before the first data item is unlocked by it; no data item is locked if it is not accessed.

- The concurrent execution of transactions is correct provided that the following rules are observed:
1. Transactions are well-formed.
2. Compatibility rules for locking are observed.
3. Each transaction does not request new locks after it has released a lock.

The last condition is expressed also by saying that the execution of every transaction proceeds through two phases: (i) Lock acquisition phase (growing phase), in which transaction must be granted locks on all data items it wants to access. (ii) Lock releasing phase (shrinking phase), in which locks are released but not requested. The end of the lock acquisition phase, when a transaction owns all the locks it will ever own, is called the commit point of the transaction.

In the 2PL algorithm it is possible to read a data item immediately after a lock on it has been granted, i.e. still in the lock acquisition phase. On the contrary, writing is possible only after the transaction has reached its commit point, that is in the lock releasing phase. More precisely, the write operation is implemented as follows. Granting a write lock is followed by a prewrite operation made in the private workspace associated with each transaction. The private workspace of a given transaction is a temporary buffer for the values of data items read from or written into the database by the transaction. The write operation to the database is performed using the contents of the workspace during the lock releasing phase, together with lock releasing. This procedure guarantees atomicity of transactions [Bern81].

Two variations of the basic 2PL algorithm have been presented and discussed in [Bern87, Bada80a]: (i) Conservative 2PL or static 2PL, in which deadlock is avoided by requiring each transaction to obtain all of its locks before any of its operations are submitted to the data manager. This is done by having each transaction predeclare its readset and writeset. (ii) Strict 2PL, almost all implementations of 2PL use this variant, in which it requires the scheduler to release all of a transaction’s locks together, when the transaction terminates. More specifically, T’s locks are released after the data manager
acknowledges the processing of T_i's commit or abort operation depending on whether T_i commits or aborts (respectively). It is worth to note that it is only necessary to hold write locks until after a transaction commits or aborts (i.e. after the data manager processes the transaction's Commit or Abort) but read locks can be released when the transaction terminates (i.e., when the scheduler receives the transaction's Commit or Abort).

There are two most common lock acquisition policies: static locking and dynamic locking. In static locking locks are acquired at once (in the beginning of the transaction). In dynamic locking locks are acquired as needed (during the execution of the transaction).

The implementation of the 2PL technique requires the introduction of a 2PL scheduler, i.e., a software module which receives lock-request and lock-release operations and executes them according to 2PL specifications. Let us assume that a logical data item X, is replicated at S sites. A transaction T can read any copy of X by first obtaining a shared lock on the copy it intends to read. However, if T wants to update X, it must first obtain an exclusive lock on all copies of X.

The implementation of a concurrency control technique of the 2PL type can force transactions to wait for the release of currently blocked data. If this wait operation is uncontrolled, a deadlock situation may arise. Consequently, deadlock detection and resolution must be associated with each 2PL concurrency control algorithm.

2.1.1.3 Algorithms Based on Locking

Distributed Two Phase Locking:

The basic way to implement 2PL in a DDB is to distribute the schedulers along with the database, placing the scheduler for data item, X, at the site where, X, is stored. Therefore, if a transaction, T, wishes to read a data item, X, the read-lock request must be sent to the local 2PL scheduler, depending on which stored copy of X is to be read. If X is to be updated, the write-lock request must be sent to all local 2PL schedulers
which contain copies of X. In this algorithms, the read-locks are requested implicitly with 
dm-reads operations and, similarly, write-locks are requested implicitly by pre-writes 
operations. If the requested lock can not be granted, the operation is placed on a waiting 
queue for desired data item.

While the write-locks are implicitly released by dm-writes operations, special 
lock-release operations are needed to release the read-locks. These lock-releases 
operations may be transmitted in parallel with the dm-writes operations, since the 
dm-writes operations signal the start of the lock releasing phase. When a lock is released, 
the operations on the waiting queue of that data item are processed in first-in-first-out 
order.

This algorithm handles redundant data correctly. If a data item X has the copies 
X₁, ..., Xₘ; if Two-phase locking is used, a transaction may read any copy of X and need 
only to obtain a read-lock on the copy of X it actually reads. However, if a transaction 
update X, then it must update all copies of X, and so must have write-locks on all copies 
of X.

An important advantage of this algorithm is that little extra communication 
between TMs and DMs is required to synchronize transactions. The only extra 
communication needed by this algorithm are the operations that release read-locks. 
However more communication is needed for deadlock detection/prevention mechanisms.

Global serializability is obtained by combining the two phase commit protocol 
with the strict two-phase locking protocol [Bern87]. This combined protocol is referred 
to as distributed 2PL.

Centralized Two Phase Locking:

Instead of distributing the 2PL schedulers with the database, it is possible to 
centralized the scheduler at a single site, known as the central site which manages all
synchronization [Geric79]. Before accessing data from any site, a transaction obtains the appropriate locks (read locks or write locks) from the central site. Once the update is computed it is broadcasted to all the sites containing the update data item. The update transaction releases the lock only after updating is finished.

One of the main disadvantages of this scheme is that the central site becomes the bottleneck of the system. Also, even if a transaction has to access local data it has to obtain a lock from central site leading to poor response time and a high communication burden.

**Primary Copy Algorithm:**

This is a variation of the centralized two phase locking algorithm [Ston79]. Here one copy of each data item is selected as its primary copy. Before accessing data from any site a transaction must obtain the appropriate lock from its primary site. In this way the problem of heavy traffic at one central node is avoided.

**Two Phase Voting Algorithm:**

This is derived from the Majority Consensus Algorithm of Thomas [Thom79]. A transaction that wants to write data item X, sends its lock request to all the nodes which has X. Upon receiving the request the site sends "lock set" or "lock blocked" to the requesting site. Once the transaction gets a majority of the "lock set" messages, it will act as if it received all "lock set" messages. Otherwise it waits for the "Lock set" messages from the sites which originally sent "lock blocked". Deadlock can occur, so deadlock detection or prevention methods must be used.
2.1.2 Timestamp Ordering Method

2.1.2.1 Timestamp Ordering Concept

Timestamp ordering (T/O) is a technique whereby the serialization order is selected a priori and transaction execution is forced to obey this order. This is in sharp contrast to 2PL, where the serialization order is induced during execution by the order in which locks are obtained.

In timestamp ordering, each transaction is assigned a unique timestamp by its TM. The TM attaches the timestamp to all dm-read and dm-write operations issued on behalf of the transaction. DMs, on the other hand, are required to process conflicting operations in timestamp order.

The first requirement of timestamps, i.e., uniqueness, can be easily satisfied in a distributed system. It is sufficient that each site adds to a locally unique timestamp its site identifier in the least significant position. Clearly, using the least significant position instead of the most significant one avoids the possibility that all timestamps which are generated by one site are greater than all timestamps which are generated by another (i.e. in the case of distributed systems, timestamps are generated by appending the site number to the timestamp generated at that site, thus ensuring uniqueness).

The second requirement, i.e., monotonicity, is more complex to satisfy [Katz79]. At each site a counter, which is steadily incremented, is used so that the transactions which receive the timestamp at the same site are correctly ordered between themselves. The counter might be simply incremented each time that a new timestamp is generated; however, the synchronization between counters at different sites would be difficult with this approach. It is possible that the counter at a site is used to generate more timestamps than the counter at a different site and hence advances faster.

Fortunately, the counters of the two sites can be kept approximately aligned by simply including in each message the value of the counter of the sending site. If a site
receives a message with a timestamp value TS which is greater than its current counter, it increments its counter to be TS+1. In this way, the counters of cooperating sites are kept approximately synchronized.

In some applications, it may be convenient to use clocks instead of counters in the implementation of timestamps. In this way timestamps reflect more closely the real time at which events occur.

Determination of ordering of events is necessary for the purposes of concurrency control. This consists of assigning to each event which occurs in the distributed system a timestamp, TS.

The aim of timestamping is to provide a simple method of sorting out incoming requests from different sources to read and write the same records [Falk85].

For any two events A and B, if A "occurred before" B, then TS(A) < TS(B). This is called the Timestamp Assignment Theorem [Bern83]. The relation "occurred before", denoted \( \rightarrow \), is defined by the following rules [Lamp78]:

1. If A and B are two events at the same site and A occurred before B, then A \( \rightarrow \) B.
2. If the event A consists in sending a message and event B consists in receiving the same message, then A \( \rightarrow \) B.
3. If A \( \rightarrow \) B and B \( \rightarrow \) C, then A \( \rightarrow \) C.

Timestamping with restarts is called basic timestamp. Instead of restarting transaction the CCA may delay the transaction. This technique is called conservative timestamp ordering and requires the reads and writes to arrive in timestamp order [Bern82].

Timestamps can be used to avoid deadlocks in database systems which use locks [Rose78] or it can be used without the locks. Timestamp ordering guarantees serialization order [Bern81].
2.1.2.2 The Basic Timestamp Algorithm

This concurrency control algorithm allows a transaction to read or write a data item x only if x had been last written by an older transaction; otherwise it rejects the operation and re-starts the transaction [Ceri84].

When a transaction $T_i$ obtains a timestamp which is smaller than the timestamp of another, already completed, transaction $T_j$, which has written data items that are needed by $T_i$, $T_i$ is aborted and re-started with a new timestamp until it obtains a timestamp which is greater than the timestamp of $T_j$.

The Basic Timestamp Algorithm applies the following rules:

1. Each transaction receives a timestamp when it is initiated at its site of origin (i.e., for each transaction $T_i$, $TS(T_i)$ is the timestamp assigned to transaction $T_i$ when it is initiated).

2. Each read or write operation which is required by a transaction has the timestamp of the transaction.

3. For each data item $x$, the largest timestamp of any dm-read operation and the largest timestamp of a dm-write operation are recorded; they are indicated as $RTM(x)$ and $WTM(x)$.

4. Let $TS$ be the timestamp of dm-read operation on data item $x$. If $TS < WTM(x)$, the dm-read operation is rejected and the issuing transaction is restarted with a new timestamp; otherwise, the dm-read is executed, and $RTM(x)$ is set to $\max(RTM(x), TS)$. Figure 2.1, is the algorithm for the read operation under basic timestamping.

5. Let $TS$ be the timestamp of a dm-write operation on data item $x$. If $TS < RTM(x)$ or $TS < WTM(x)$, then the operation is rejected and the issuing transaction is re-started; otherwise, the write is executed, and $WTM(x)$ is set to $TS$. Figure 2.2 is the algorithm for the write operation under basic timestamping.
Figure 2.1 Read Operation Using Basic Timestamping

begin
    T; attempts a read operation on data item x
    if x has been updated by a younger transaction
        i.e. TS(T;) < WTM(x)
        then reject read operation and restart T; with a new timestamp.
        else accept read operation and
            set RTM(x) = max(RTM(x), TS(T;))
    end-if
end

Figure 2.2 Write Operation Using Basic Timestamping

begin
    T; attempts to update (write) data item x
    if x has been read or written by a younger transaction
        i.e. TS(T;) < RTM(x) or TS(T;) < WTM(x)
        then reject write operation and restart T; with a new timestamp
        else accept write operation and
            set WTM(x) = TS(T;)
    end-if
end

Rules 4 and 5 ensure that conflicting operations are executed in timestamp order at all sites; hence the timestamp order is a total order and the executions produced by this algorithm are correct.

An interesting feature of the Basic Timestamp algorithm is that it is deadlock free, because transactions are never blocked; if a transaction cannot execute an operation, it is re-started. Waiting would be meaningless in this approach anyway. If an operation cannot be granted, this does not depend on the fact that another transaction is momentarily operating on the same data item, but instead depends on a permanent situation, since data items never decrease their timestamps. However, deadlock freedom is obtained at the cost of re-starting transactions, rather than making them wait.
The basic rules which have been described are sufficient to ensure the serializability of transactions; however, they need to be integrated with two-phase commitment to ensure atomicity. Two-Phase commitment requires that there be a time interval during which all the agents of a transaction are capable of aborting or committing. With a locking mechanism this is possible, because all exclusive locks are held by a transaction until the end of commitment; (and no transaction can read the data which has been written by a not yet committed transaction). With the timestamp mechanism a different solution is needed: instead of exclusive locks, pre-writes are used. Pre-writes are issued by transactions instead of write operations; they are buffered and not applied directly to the database. Only when the transaction is committed, are the corresponding write operations applied to the database. In this way, if the pre-writes of a transaction have been accepted (buffered), at transaction commit the corresponding writes will not be rejected. Buffering of an operation means that the operation is neither executed, nor rejected; instead it is recorded together with its timestamp for subsequent execution, and it is ensured that this execution will be possible at a later time.

In order to integrate the basic timestamp algorithm and two-phase commitment, the above rules 4 and 5 are substituted by the following rules 4, 5 and 6.

4. Let TS be the timestamp of a pre-write operation P\textsubscript{j} on data item x. If TS < RTM(x) or TS < WTM(x), then the operation is rejected and the issuing transaction is re-started; otherwise, the pre-write P\textsubscript{j} and its timestamp TS are buffered.

5. Let TS be the timestamp of a read operation R\textsubscript{i} on data item x. If TS < WTM(x), the operation is rejected. However, if TS > WTM(x), then R\textsubscript{i} is executed only if there is no pre-write operation P(x) pending on data item x having a timestamp TS(P) < TS. If there is one (or more) pre-write operation P(x) with TS(P) < TS, R\textsubscript{i} is buffered until the transaction which has issued P(x) commits. The reason
why $R_i$ is buffered is that the write operation $W(x)$ corresponding to the pre-write $P(x)$ cannot be rejected; therefore we must avoid $TS(W) < RTM(x)$. But $TS(W) = TS(P)$, because they are issued by the same transaction; applying $R_i$ is avoided since the value of $RTM(x)$ would be set equal to the TS, thus making $W(x)$ impossible. The read operation $R_i$ will be executed and eliminated from the buffer when no more pre-writes with a smaller timestamp than $R_i$ are pending on $x$.

6. Let $TS$ be the timestamp of a write operation $W_i$ on data item $x$. This operation is never rejected; however, it is possibly buffered if there is a pre-write operation $P(x)$ with a timestamp $TS(P) < TS$, for the same reason which has been stated for buffering read operations. $W_i$ will be executed and eliminated from the buffer when all pre-writes with smaller timestamps have been eliminated from the buffer.

The use of pre-writes is equivalent to applying exclusive locks on data items for the time interval between the pre-write and the commitment (write) or abort of the issuing transaction.

The "ignore obsolete write" rule, Rule 5 of the Basic Timestamp Algorithm can be modified in the following way: if the timestamp of a write operation $W_i(x)$ is smaller than the write timestamp $WTM(x)$ of the data item $x$, it is possible to ignore the operation, instead of rejecting the operation and re-starting the transaction. The reason why this simplified rule works correctly is that if $W_i$ and $W_j$ are two write operations such that $TS(W_i) < TS(W_j)$, the execution of $W_i$ followed by $W_j$ is equivalent to the execution of $W_j$ alone. If therefore $W_i$ is issued after $W_j$ by ignoring it the same result is obtained as if it were executed before $W_j$.

This situation can only occur if transaction $T_j$ did not read data item $x$; otherwise $W_i$ would be rejected, because $TS(W_i) < RTM(x)$. This fact confirms an intuitive
argument: if a transaction \( T_i \) writes a data item \( x \) without reading it previously, then the value of \( x \) after \( W_j(x) \) is independent of the previous history (Previous writes) of \( x \). However, in general, transactions which write a data item do also read it [Ceri84].

In the presentation of the basic timestamp algorithm with two-phase commitment we use the following notation. A queue associated with a data item \( x \) where read, prewrite and write requests are buffered is denoted by \( \text{BufQ}(x) \). The smallest timestamp of transactions whose read, prewrite or write requests are buffered in \( \text{BufQ}(x) \) is denoted by \( \min\text{-R-ts}(x) \), \( \min\text{-P-ts}(x) \) and \( \min\text{-W-ts}(x) \) respectively. The read, prewrite and write procedures of the basic timestamp algorithm with two phase commitment are the following [Bern80, 80a]:

**Procedure 2PC - Read** \((T_i, x)\);

\[
\text{begin} \\
\quad \text{if } \text{TS}(T_i) < \text{WTM}(x) \text{ then} \\
\qquad < \text{abort } T_i \text{ and restart it with a new timestamp} > \\
\quad \text{else if } \text{TS}(T_i) > \min\text{-P-ts}(x) \text{ then} \\
\qquad < \text{place } T_i \text{'s read request in the } \text{BufQ}(x) > \\
\quad \text{else begin} \\
\qquad < \text{read } x > ; \\
\quad \text{end} \\
\text{RTM}(x) \leftarrow \max\{\text{TS}(T_i), \text{RTM}(x)\} \\
\text{end}
\]

**Procedure 2PC - Pre Write** \((T_i, x)\);

\[
\text{begin} \\
\quad \text{if } \text{TS}(T_i) < \text{RTM}(x) \text{ or } \text{TS}(T_i) < \text{WTM}(x) \text{ then} \\
\qquad < \text{abort } T_i \text{ and restart it with a new timestamp} > \\
\quad \text{else } < \text{place } T_i \text{'s prewrite request in the } \text{BufQ}(x) > \\
\text{end}
\]

**Procedure 2PC-Write** \((T_i, x)\);

\[
\text{begin} \\
\quad \text{if } \text{TS}(T_i) > \min\text{-R-ts}(x) \text{ or } \text{TS}(T_i) > \min\text{-P-ts}(x) \text{ then} \\
\qquad < \text{place } T_i \text{'s write request in the } \text{BufQ}(x) > \\
\quad \text{else begin} \\
\qquad < \text{call Realize-write } (T_i, x) > ; \\
\qquad \text{WTM}(x) \leftarrow \text{TS}(T_i) \\
\quad \text{end} \\
\text{end}
\]

**Procedure Realize-write** perform DDB updates. It updates \( x \) to a new value and removes
Tj's prewrite request from BufQ(x). If the value of min-P-ts(x) increases after the removal of Tj's prewrite request from BufQ(x), a procedure is initiated to test BufQ(x). This procedure determines if the increase of the value of min-P-ts(x) has not unblocked any read or write requests of another transaction, say Tj, for which the condition TS(Tj) < min-P-ts (x) is now satisfied. Unblocking these requests may trigger another update of min-R-ts(x) and min-P-ts(x), and consequently another test of BufQ(x). It is worth to note that incorporating the two-phase commitment into the basic timestamp ordering method increase its complexity and reduces the concurrency degree.

2.1.2.3 Timestamp Based Algorithms:

Thomas Majority Consensus Algorithm:

This algorithm [Thom79] assumes that every data item is stored at all the sites, i.e. fully redundant. A timestamp of the last transaction that updated that data is stored along with the data. A transaction is assigned a unique timestamp when it begins executing. Transactions execute in two stages. In the first stage each transaction is executed at the local site. A data item that is written is recorded in an update list. At the end of execution this update list is appended to the list of data items read and their timestamps. In the second stage the update list is sent to all the sites. Each of these sites votes (yes or no) and sends it to the home site. If the vote is yes then that update list is said to be pending at that site. When the home site receives a majority of yes or no votes it informs all the sites. If the vote is yes then the update is committed at all the nodes. If the vote is no the update is discarded and the transaction is restarted.

A site votes "no" if any of the data item's timestamp in the update list is different from the local timestamp for that data. Otherwise, if there is no read-write conflict with any of the pending updates it votes "yes". If there is a read-write conflict then it votes pass if the update's timestamp is later than all the conflicting pending updates. If a local
timestamp is more recent than the update list the update is put in a waiting queue for later consideration.

**Weight Voting Algorithm:**

Instead of each copy having one vote, different copies can be assigned different votes. The weights are assigned so as to improve performance (reliability and resiliency).

**Thomas Write Rule:**

If the transaction is just writing data (without any read) then it is always accepted. When a transaction is received if its timestamp is later than the timestamp on the data then update is done otherwise it is just discarded (without informing the transaction). This results in fewer rejections of update transactions and also the communication overhead is reduced.

**SDD-1 Approach:**

The main principle behind their approach [Bern80b] is not all transactions require synchronization as strong as locking. This method is based on a conflict analysis of the transactions. Transactions are grouped into different classes for the purpose of determining the level of synchronization required to avoid conflict and guarantee a serializable execution of transactions. Four different protocols were developed and each of these protocols allows different amount of concurrency. When a transaction is initiated by the user it will be associated with one of the classes and a corresponding protocol will be used.
2.1.3 Optimistic Method

2.1.3.1 Optimistic Concept

The basic idea of optimistic methods is the following: instead of suspending or rejecting conflicting operations, like in 2-Phase Locking and Timestamping, always execute a transaction to completion. However, the write operations issued by transactions are performed on local copies of the data. Only at the end of the transaction, if a validation test is passed by the transaction, are the writes applied to the database. If the validation test is not passed, the transaction is re-started. The validation test verifies if the execution of the transaction is serializable. In order to perform the test, some information about the execution of the transaction must be retained until the validation is performed.

As a transaction executes, information about the set of items read, written, and created by the transaction is collected. At the end, the transaction is validated to determine whether or not to commit the transaction. Values created by an uncommitted transaction are not available to other transactions, and only after the transaction has committed are the values made globally available.

The optimistic approach is based on the assumption that conflicts are rare and therefore most transactions will pass the test. By processing operations without concurrency control overhead, a transaction is not delayed during its execution.

The idea behind the optimistic approach is quite simple, and may be summarized as follows [Kung81]:

1. Since reading a value can never cause a loss of integrity, reads are completely unrestricted (however, returning a result from a query is considered to be equivalent to a write, and so is subject to a validation).

2. Writes are severely restricted. It is required that any transaction consists of three phases: a read phase, a validation phase, and a possible write phase. In the case
of a query, it must be determined that the result the query would return is actually correct.

This approach is quite efficient, provided that the probability of the conflict between two transactions is very small. The chance of interaction between two transactions executing at about the same time will be small provided each accesses only a tiny fraction of the database, i.e., the lock granularity is small. If this situation holds, there is substantial benefit in using the optimistic approach, as no time will be wasted requesting or waiting for locks. The time wasted re-doing transactions that have to be aborted may well be much less than the time spent waiting for locks or undoing deadlocks [Ullm82].

2.1.3.2 Algorithms Based on Optimistic Approach

Kung-Robinson Optimistic Algorithm:

The execution of a transaction according to the kung - Robinson algorithm [Kung81] Proceeds in three phases:

1. Read Phase: During this phase, a transaction reads data items from the database, performs computations, and determines new values for the data items of its write-set; however these values are not written in the database. It should be noted that the read phase contains almost the whole execution of the transaction.

2. Validation Phase: During this phase, a test is performed to see whether the application of the updates to the database which have been computed by the transaction would cause a loss of consistency or not.

3. Write Phase: During this phase, the updates are applied to the database if the validation phase has returned a positive result; otherwise the transaction is re-started. A transaction is re-started if it finds that any item that it reads has been written by another transaction which committed during its life-time.
If validation does fail, the transaction will be backed-up and will start over again as a new transaction. Thus a transaction will have a write phase only if the preceding validation succeeds.

If this method is first considered in a centralized database, the following scheme is obtained:

Each transaction receives a timestamp during its execution, however, not at its initiation, since timestamps are only needed during the validation phase. To assign timestamps, a global integer counter (Transaction Number Counter, TNC) is maintained. When a transaction number is needed, this counter is incremented and the resulting value is returned. Transaction numbers must be assigned somewhere before validation, since the validation conditions of the validation phase require knowledge of the transaction number of the transaction being validated. Transaction numbers might be assigned at the beginning of the read phase; however this is not optimistic for the following reason. In the case, where two transactions $T_1$ and $T_2$ start at roughly the same time, and assigned transaction numbers $n$ and $n+1$ respectively, even if $T_2$ completes its read phase much earlier than $T_1$, before being validated $T_2$ must wait for the completion of the read phase of $T_1$, since the validation of $T_2$ in this case relies on the knowledge of the write-set of $T_1$.

In an optimistic approach, it is desirable for transactions to be validated immediately, if at all possible (in order to improve response time). For these and similar considerations transaction numbers are assigned at the end of the read phase.

When the validation fails, the transaction is aborted and re-started, receiving a new transaction number at the completion of the read phase. The case in which validation repeatedly fails is expected to happen under the optimistic assumptions. In such a case, if the concurrency control detects a starving transaction (this could be detected by keeping track of number of times validation for a given transaction fails), the transaction is
enabled to run to completion. This is equivalent to write-locking the entire database.

The validation procedure of a transaction $T_j$ is based on checking that the schedule produced by already committed transactions, transactions which are in their validation phase when $T_j$ is being validated, and $T_j$ itself is equivalent to a serial schedule in timestamp order.

The validation condition for a transaction $T_j$ with timestamp $TS(T_j)$ requires that, for all transactions $T_i$ with $TS(T_i) < TS(T_j)$, one of the following three conditions hold:

**Condition 1:** $T_i$ completes its write phase before $T_j$ starts its read phase. This effectively means that $T_i$ has finished before $T_j$ begins.

**Condition 2:** The write-set of $T_i$ does not intersect the read-set of $T_j$, and $T_i$ completes its write phase before $T_j$ starts its write phase. This means that the set of data items updated by $T_i$ (write-set($T_i$)) can not affect the set of data items read by $T_j$ (read-set($T_j$)) and that $T_i$ can not overwrite $T_j$ because it will have finish writing before $T_j$.

**Condition 3:** The write-set of $T_i$ does not intersect the read-set or the write-set of $T_j$ and $T_i$ completes its read phase before $T_j$ completes its read phase. This ensures that $T_i$ does not affect either the read or write phase of $T_j$.

In order to transform the above rules into an algorithm, during the execution of $T_j$ the following information is recorded:

1. The read and write-sets of $T_j$
2. The value of TNC when $T_j$ is started; this called START($T_j$)
3. The value of TNC when $T_j$ finished its read phase; this is called FINISH($T_j$).

$START(T_j)$ and $FINISH(T_j)$ are therefore two local variables of $T_j$ containing timestamp values that are needed for performing validation and that will be discarded after $T_j$ is terminated completely; the definitive, unique timestamp $TS(T_j)$ is assigned to
\( T_j \) only after the write phase if the validation succeeds. TNC is incremented only when the definitive timestamp is assigned, so that subsequent transactions will receive a greater timestamp. Figure 2.3a shows the meaning of \( \text{START}(T_j) \), \( \text{FINISH}(T_j) \), and \( \text{TS}(T_j) \).

The algorithm for the validation of a transaction \( T_j \) by considering each one of the above conditions separately as shown in Figure 2.3b, c, d:

1. For all transactions \( T_i \) such that \( \text{TS}(T_i) < \text{START}(T_j) \), nothing has to be checked. Therefore the validation algorithm has to consider only transactions \( T_i \) such that \( \text{TS}(T_i) > \text{START}(T_j) \) (Figure. 2.3.b).

2. The transactions \( T_i \) which have terminated their write phase during the read phase of \( T_j \) are identified by the validation algorithm by checking whether \( \text{START}(T_j) < \text{TS}(T_i) < \text{FINISH}(T_j) \). For all these transactions the above condition 2 is checked (Figure. 2.3.c).

3. The transactions \( T_i \) such that \( \text{FINISH}(T_j) < \text{FINISH}(T_i) \) are identified by keeping track of all transactions which have not yet terminated execution, and are called \textbf{active} transactions. For all these transactions the above condition 3 is checked (Figure. 2.3.d).
Figure 2.3 Validation conditions for optimistic concurrency control
When a transaction $T_j$ is ready to validate, having reached the end of its read phase (timestamp = $\text{FINISH}(T_j)$), it must check that one of the these conditions listed above holds. Figure 2.4 illustrates this validation process for a centralized database.

**Figure 2.4** Validation Using Optimistic Concurrency Control for Centralized DBMS.

begin
Validate transaction $T_j$ (transaction timestamp $TS(T_j)$) against all other older, committed transactions $T_i$
for all $T_i$ where $TS(T_i) < TS(T_j)$ do
begin
  Condition 1: $T_j$ has completed its write phase before $T_j$ starts its read phase
  if $TS(T_i) < \text{START}(T_j)$
    then return success
  end-if
  Condition 2: readset of $T_j$ and writeset of $T_j$ are disjoint, and $T_j$ has completed its write phase before $T_j$ starts its read phase
  if $\text{writeset}(T_j) \cap \text{readset}(T_j) = \emptyset$
    and $TS(T_i) < \text{FINISH}(T_j)$
    then return success
  end-if
  Condition 3: writeset of $T_j$ does not intersect with either the readset of $T$, of the writeset of $T$, and $T_i$ completes its read phase before $T_j$ completes its read phase
  if $\text{writeset}(T_j) \cap (\text{readset}(T_i) \cup \text{writeset}(T_i)) = \emptyset$
    and $\text{FINISH}(T_i) < \text{FINISH}(T_j)$
    then return success
  end-if
All conditions fail, so validation fails
return failure
end-do
No older transactions; return success

This approach has a possibility of indefinite postponement, but it can be avoided by counting the number of times a transaction is not validated and after some limit it is given high priority during validation.

**G. Schlageter Algorithm:**

An improvement on the above algorithm has been proposed by
G. Schlageter [Schl81]. In this algorithm READ transactions are not considered for validation. They are always accepted. All the responsibility for validation is left to the update. An update transaction waits for all the conflicting read transactions to complete before it begins validation. There is an additional possibility of indefinite wait for the update transactions due to the reader transactions. But it can be avoided by limiting the number of new readers allowed while updates are waiting. This approach results in fewer of updates.

Ceri's and Owicki Algorithm:

Extension of the optimistic algorithm [Kung81] to distributed database was proposed by Ceri and Owicki [Ceri82]. The validation phase was changed to incorporate the distributed nature of the transactions.

In the case of a distributed database, each distributed transaction is constituted by subtransactions. $T_i^j$ becomes the subtransaction which executes at site $j$ for transaction $T_i$. The validation method is extended to the distributed environment in the following way [Ceri84]:

1. Each subtransaction of the same transaction receives the same global, unique transaction identifier.
2. Each site performs a local validation of each local subtransaction.
3. For each subtransaction $T_i^j$, the happened before set, HB($T_i^j$) is built, which contains the identifiers of those global transactions which precede $T_i^j$ in the serialization order at site $j$. The information which is necessary for building the HB-set is collected during the local validation phase.
4. Then, the real global validation phase is performed. Global validation is based on a simple (but pessimistic) idea: a subtransaction is considered valid only if all global transactions which belong to its HB-set have either committed or aborted.
If it is not yet known for some of these transactions whether they have committed or aborted, the validation of $T_i^j$ is suspended. This waiting can cause a deadlock; hence, after a time-out, if the subtransaction cannot be validated, it is aborted. Therefore, this method is effective only if most transactions do not require waiting during global validation. This is shown in Figure 2.5.

5. Write phase: 2-phase commitment can be used. When a subtransaction has finished its global validation (positively or negatively), the corresponding message is sent to the transaction's master site. If all validations are positive, the transaction is committed, otherwise, it is aborted.

The fundamental rule of global validation is rule-4. The following example provides a better understanding of this rule [Ceri84]. Two transactions $T_i$ and $T_j$ have executed subtransactions at two sites h and k in such a way that the schedule at site h is equivalent to a serial schedule $T_p,T_i$ and the schedule at site k is equivalent to the serial schedule $T_p,T_j$. This means that at both sites the following situation exists:

Site-h: $T_i^h$ validated; $T_j^h$ waiting with HB($T_j^h$) = \{T_i\}

Site-k: $T_j^k$ validated; $T_i^k$ waiting with HB($T_i^k$) = \{T_j\}

This is a typical deadlock situation. The time-out method will cause at least one transaction to be aborted, so that the inconsistent execution cannot be produced.

Optimistic methods are assumed to be convenient for systems where transactions have only few conflicts. This is especially true in applications where most transactions are read-only. A read-only transaction has no write phase and no write-set, so that its validation consists only in checking condition-2 [Ceri84].
Figure 2.5 Distributed Validation Under Conservative Timestamping.

**Begin**

Phase 1: All the local sub-transactions of $T_j$ are validated independently as for centralized protocol.

Let the local schedule at site $i$, $S^i$ be

$T_{i1}, T_{i2}, ..., T_{ij}, ..., T_{im}$

Note that this schedule contains local transactions as well as sub-transactions of global transactions. If any $S^i$ is not serializable (validation fails) then global transaction $T_j$ is aborted.

Phase 2: Assuming that all $S^i$ are serializable, global validation of $T_j$ proceeds as follows.

Check that all sub-transactions of global transactions which precede $T_j$, namely $T_{i1}, T_{i2}, ..., T_{jn}$, have finished, i.e. committed or aborted, otherwise suspend validation of $T_j$. $T_j$ waits until either these sub-transaction finish, or $T_j$ times out and is therefore aborted.

**End**

### 2.1.4 Hybrid Method

From the comparative analysis of the basic concurrency control methods discussed so far, their advantages and drawbacks are to a large extent complementary. Therefore, it seems only natural to try to integrate them and design hybrid methods. Algorithms exist most commonly use 2PL and timestamp ordering [Bern80, 81, 82]. The "Wait-Die" and "Wound-Wait" protocols which we have discussed in sections 1.5, are deadlock avoidance techniques which use locking with timestamps and prevent cycles from occuring in the wait-for-graph.

### 2.2 Transaction Commit

A distributed transaction $T$ accesses data items located at two or more sites. A commit operation concerns all sites involved in the processing of $T$. Consequently, the TM of $T$'s "home site" the site where it originated, should pass the commit operation of $T$ to all sites where $T$ accessed data items. The same is true for abort.

In section 2.1.1 2PL method that prevent interaction between transactions were described. Even using this algorithm inconsistency in DDBS can arise if transactions are
allowed to commit at some sites while at other sites they are aborted due to site failure or communication links failure. To ensure that either the transaction is committed at all the sites or aborted at all sites an algorithm called atomic commitment protocol (ACP) is used. The simplest and the most popular ACP is the two phase commit (2PC) protocol [Gray79].

2.2.1 Two Phase Commit Protocol

Two phase commit ensures that either the transaction is committed at all the sites or aborted at all sites. This property, that transactions commit at all sites or none, is referred to as atomic commit. The site, where the transaction originated, coordinates with all other sites. This transaction originating site becomes the commit coordinator (or commit master) and all the other sites which participate in the committing of the transaction become participants. When the transaction is ready to commit the 2PC protocol is started. Assuming no failure 2PC protocol goes as follows:

In the first phase the coordinator writes a "prepare" record in the log and sends a PREPARE message to all the participants. Upon the receipt of the PREPARE message, all the participating sites record all the values to be updated in stable storage, send a READY message, if it is ready to commit, otherwise it sends an ABORT message. The coordinator decides whether to commit or abort the transaction after receiving an answer from the participants. If all the messages received were READY then the coordinator decides to commit the transaction, instead if some of the messages were ABORT then it decides to abort the transaction.

In the second phase the coordinator records either Global-Commit or Global-Abort on its stable storage and accordingly sends either a COMMIT or an ABORT message to all the participants. Participants then send an ACK message to the coordinator. The coordinator receives this message and records "complete" in its log.
By applying this protocol the transaction is committed at all sites or aborted at all sites. It can be shown that the above protocol is resilient to almost all failures. Detail of 2PC protocol assuming no failure is given in Figure 2.6.

Figure 2.6 Basic 2-Phase-Commitment Protocol Assuming No Failure

**Coordinator:** Write "prepare" record in the log;  
Send PREPARE message

**Participant:** Wait for PREPARE message;  
If the participant is willing to commit then  
begin  
Write subtransaction's records in the log;  
Write "ready" record in the log;  
Send READY answer message to coordinator  
end  
else begin  
Write "abort" record in the log;  
Send ABORT answer message to coordinator  
end

**Coordinator:** Wait for ANSWER message (READY or ABORT) from all participants;  
If some answer message is ABORT then  
begin  
Write "global-abort" record in the log;  
Send ABORT command message to all participants  
end  
else (* all answers arrived and were READY*)  
begin  
Write "global-commit" record in the log;  
Send COMMIT command message to all participants  
end

**Participant:** Wait for command message;  
Write "abort" or "commit" record in the log;  
Send the ACK message to coordinator;  
Execute command

**Coordinator:** Wait for ACK messages from all participants;  
Write "complete" record in the log