Faculty of Computer Science and Engineering
Ho Chi Minh City University of Technology

Chapter 5: Transaction processing and concurrency control

Graduate Programme in Computer Science

Dr. Vo Thi Ngoc Chau
(chauvtn@cse.hcmut.edu.vn)

Semester 2 – 2011-2012

Main references

[1] R. Elmasri, S. R. Navathe, *Fundamentals of Database Systems- 4th Edition*, Pearson- Addison Wesley, 2003.

[2] H. G. Molina, J. D. Ullman, J. Widom, *Database System Implementation*, Prentice-Hall, 2000.

[3] H. G. Molina, J. D. Ullman, J. Widom, *Database Systems: The Complete Book,* Prentice-Hall, 2002

[4] A. Silberschatz, H. F. Korth, S. Sudarshan, *Database System Concepts –3rd Edition*, McGraw-Hill, 1999.

[5] R. Ramakrishnan, J. Gehrke, *Database Management Systems- 2rd Edition,* McGraw-Hill.

[6] M. P. Papazoglou, S. Spaccapietra, Z. Tari, *Advances in Object-Oriented Data Modeling*, MIT Press, 2000.

[7]. G. Simsion, *Data Modeling: Theory and Practice*, Technics Publications, LLC, 2007.

Content

Chapter 5: Transaction processing and concurrency control

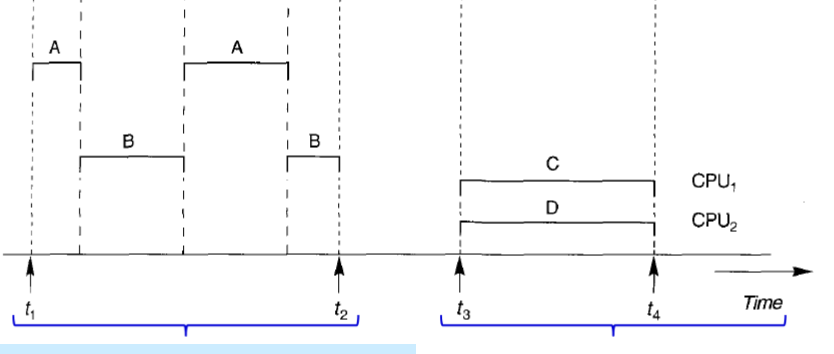
5.1. An overview of transaction processing

- Single-user vs. multi-user database systems
  - Concurrent execution of transactions in a multi-user system
  - Recovery from transaction failures when transactions fail while executing

→ The number of users who can use the system concurrently-that is, at the same time
  - Single-user if at most one user at a time can use the system
  - Multiuser if many users can use the system - and hence access the database - concurrently

Figure 17.1. Interleaved processing versus parallel processing of concurrent transactions.

[1], pp. 553.



Concurrent execution of processes is actually interleaved. Interleaving keeps the CPU busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. Interleaving also prevents a long process from delaying other processes.

If the computer system has multiple hardware processors (CPUs), *parallel processing* of multiple processes is possible.

5.1. An overview of transaction processing

- Concepts
  - Transaction
    - Commit point
    - Transaction states
  - Database access operations
- Issues related to transaction processing
  - Concurrency control
  - Recovery
    - System log

5.1. An overview of transaction processing

- Transaction
  - An executing program that forms a logical unit of database processing
    - Including one or more database access operations
      - Insertion, deletion, modification, or retrieval operations
      - Embedded within an application program or specified interactively via a high-level query language such as SQL
  - The transaction boundaries: explicit begin transaction and end transaction statements in an application program
    - All database access operations between the two are considered as forming one transaction.

5.1. An overview of transaction processing

- Basic database access operations that a transaction can include:
  - **read_item(X)**: reads a database item named X into a program variable. To simplify our notation, we assume that *the program variable is also named* X.
  - **write_item(X)**: writes the value of program variable X into the database item named X.

  → *Note*: the basic unit of data transfer from disk to main memory is one *block*.

5.1. An overview of transaction processing

(a) T_1	(b) T_2
read_item (X);	read_item (X);
X:=X-N;	X:=X+M;
write_item (X);	write_item (X);
read_item (Y);	
Y:=Y+N;	
write_item (Y);	

Figure 17.2. Two sample transactions. (a) Transaction T1. (b) Transaction T2. [1], PP. 555.

Transactions submitted by the various users may execute concurrently and may access and update the same database items. If this concurrent execution is uncontrolled, it may lead to problems, such as an inconsistent database.
→ *Concurrency control* and *recovery* mechanisms are mainly concerned with the database access commands in a transaction.

5.1. An overview of transaction processing

- Commit point of a transaction
  - A transaction T reaches its **commit point** when all its operations that access the database have been executed successfully *and* the effect of all the transaction operations on the database have been recorded in the log.
  - Beyond the commit point, the transaction is said to be **committed,** and its effect is assumed to be *permanently recorded* in the database.

11

5.1. An overview of transaction processing

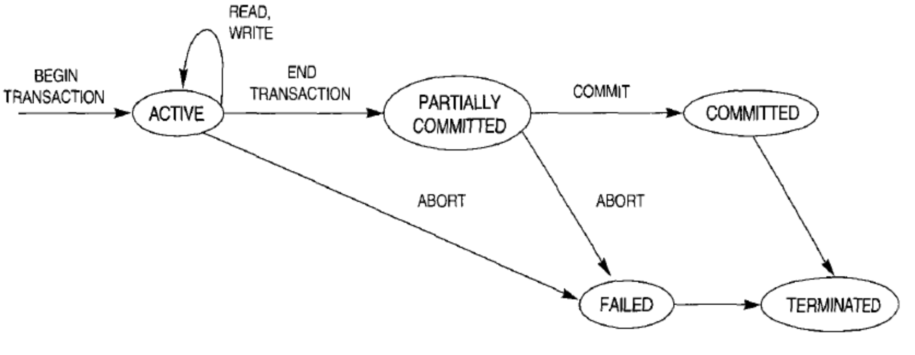


Figure 17.4. State transition diagram illustrating the states for transaction execution. [1], pp. 560.

12

5.1. An overview of transaction processing

- Transaction states
  - *Active state*: a transaction goes into an active state immediately after it starts execution, where it can issue READ and WRITE operations.
  - *Partially committed*: when the transaction ends, some recovery protocols need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently (usually by recording changes in the system log).
  - *Committed*: once this check is successful, the transaction is said to have reached its commit point. Once a transaction is committed, it has concluded its execution successfully and all its changes must be recorded permanently in the database.
  - *Failed*: if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. Failed or aborted transactions may be *restarted* later - either automatically or after being resubmitted by the user-as brand new transactions.
  - *Terminated*: the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates.

5.1. An overview of transaction processing

- Why concurrency control is needed

→ The problems encountered with the transactions running concurrently if uncontrolled

  - The lost update problem
  - The temporary update (or dirty read) problem
  - The incorrect summary problem
  - The unrepeatable problem
  - The phantom problem

5.1. An overview of transaction processing

- The *lost update* problem
  - This problem occurs when two transactions that access the same database items have their operations interleaved in a way that makes the value of some database items incorrect.

Time

T_1	T_2
read_item(X); X:=X-N;	
	read_item(X); X:=X+M;
write_item(X); read_item(Y);	
	write_item(X);
Y:=Y+N; write_item(Y);	

Item X has an incorrect value because its update by T_1 is "lost" (overwritten)

5.1. An overview of transaction processing

- The ***temporary update*** (or ***dirty read***) problem
  - This problem occurs when one transaction updates a database item and then the transaction fails for some reason. The updated item is accessed by another transaction before it is changed. The updated item is accessed by another transaction before it is changed.

Time

T_1	T_2

read_item(X);
X:=X-N;
write_item(X);	

	read_item(X);
	X:=X+M;
	write_item(X);

read_item(Y);	

Transaction T_1 fails and must change the value of X back to its old value; meanwhile T_2 has read the "temporary" incorrect value of X.

T_1	T_3	

	sum:=0;
	read_item(*A*);
	sum:=*sum*+*A*;
	⋮	

read_item(*X*);
X:=*X*-N;
write_item(*X*);		

	read_item(*X*);
	sum:=*sum*+*X*;
	read_item(*Y*);
	sum:=*sum*+*Y*;
		← T_3 reads *X* after *N* is subtracted and reads *Y* before *N* is added; a wrong summary is the result (off by *N*).

read_item(*Y*);
Y:=*Y*+*N*;
write_item(*Y*);		

- The *incorrect summary* problem
  - If one transaction is calculating an aggregate summary function on a number of records while other transactions are updating some of these records, the aggregate function may calculate some values before they are updated and others after they are updated.

5.1. An overview of transaction processing

- The *unrepeatable (nonrepeatable)* problem
  - A transaction T reads an item twice and the item is changed by another transaction T' between the two reads. Hence, T receives *different values* for its two reads of the same item.

5.1. An overview of transaction processing

- The *phantom* problem
  - A transaction T1 may read a set of rows from a table, perhaps based on some condition specified in the SQL WHERE-clause. Now suppose that a transaction T2 inserts a new row that also satisfies the WHERE-clause condition used in T1, into the table used by T1. If T1 is repeated, then T1 will see a phantom, a row that previously did not exist.

5.1. An overview of transaction processing

- Why recovery is needed
  - Whenever a transaction is submitted to a DBMS for execution, the system is responsible for making sure:
    - either (1) all the operations in the transaction are completed successfully and their effect is recorded permanently in the database,
    - or (2) the transaction has no effect whatsoever on the database or on any other transactions.
  - The DBMS must not permit some operations of a transaction T to be applied to the database while other operations of T are not.
  - This may happen if a transaction fails after executing some of its operations but before executing all of them.

5.1. An overview of transaction processing

- Several possible reasons for a transaction to fail in the middle of execution:
  - 1. A *computer failure (system crash)*
  - 2. A *transaction* or *system error*
  - 3*. Local errors* or *exception conditions detected by the transaction*
  - 4*. Concurrency control enforcement*
  - 5*. Disk failure*
  - 6*. Physical problems and catastrophes*
  - → Recovery and backup

5.1. An overview of transaction processing

- Recovery
  - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
  - The recovery manager keeps track of the following operations:
    - BEGIN_TRANSACTION
    - READ/WRITE
    - END_TRANSACTION
    - COMMIT_TRANSACTION
    - ROLLBACK (ABORT)

5.1. An overview of transaction processing

- Recovery log (System log)
  - To be able to recover from failures that affect transactions, the system maintains a log to keep track of all transaction operations that affect the values of database items.
  - The log is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
  - The log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.
  - The log contains entries - called log records.
    - In these entries, T refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction.

5.1. An overview of transaction processing

- Log records where T refers to a unique transaction-id that is generated automatically by the system and is used to identify each transaction:
  - [start_transaction, T]
  - [write_item, T, X, old_value, new_value]
  - [read_item, T, X]
  - [commit, T]
  - [abort, T]

5.1. An overview of transaction processing

- Recovery from a transaction failure
  - Undoing or redoing transaction operations individually from the log
    - → If the system crashes, recover to a consistent database state by examining the log and using a recovery technique.
    - → Undoing the effect of these WRITE operations of a transaction T by ***tracing backward*** through the log and resetting all items changed by a WRITE operation of T to their old_values
    - → Redoing the operations of a transaction T by ***tracing forward*** through the log and setting all items changed by a WRITE operation of T to their new_values

5.2. Desirable properties of transactions

- Transactions possess the ACID properties enforced by the concurrency control and recovery methods.
  - Atomicity
    - A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
  - Consistency preservation
    - A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.
  - Isolation
    - A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
  - Durability (or permanency)
    - The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

5.3. Characterizing transaction schedules based on recoverability

- Schedule (or history)
  - The *order of execution of operations* from the various transactions when transactions are executing concurrently in an interleaved fashion
  - Types of schedules
    - Based on *recoverability*
      - Schedules facilitate recovery from transaction failures occur.
    - Based on *serializability*
      - Schedules with the interference of participating transactions

5.3. Characterizing transaction schedules based on recoverability

- Constraint on schedules
  - Consider a schedule (or history) *S* of *n* transactions T_1, T_2, …, T_n
  - For each transaction T_i that participates in *S*, the operations of T_i in *S* must appear in the same order in which they occur in T_i. However, operations from other transactions T_j can be interleaved with the operations of T_i in *S*.

5.3. Characterizing transaction schedules based on recoverability

TIME

T1	T2	Interleaved Execution
read_item(X);		$r_1(X)$
X:=X-N;		
write_item(X);		$w_1(X)$
	read_item(X);	$r_2(X)$
	X:=X+M;	
	write_item(X);	$w_2(X)$
read_item(Y);		$r_1(Y)$
abort		a_1
	abort	a_2

Schedule S_b: $r_1(X)$; $w_1(X)$; $r_2(X)$; $w_2(X)$; $r_1(Y)$; a_1; a_2

$r_1(X)$ = T_1 reads X.
$w_1(X)$ = T_1 writes X.
a_1 = abort of T_1
c_3 = commit of T_3

29

5.3. Characterizing transaction schedules based on recoverability

$$S_a: r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); w_1(Y);$$

Schedule S_a

write (w) operation of **transaction T_1**

Time

T_1	T_2
read_item(*X*); *X*:=*X*-*N*;	
	read_item(*X*); *X*:=*X*+*M*;
write_item(*X*); read_item(*Y*);	
	write_item(*X*);
Y:=*Y*+*N*; write_item(*Y*);	

30

5.3. Characterizing transaction schedules based on recoverability

- *Recoverable* schedules
  - *Recoverable* schedule: once a transaction T is committed, it should never be necessary to roll back T.
    - No transaction T in a schedule ***commits*** until all transactions T' that have written an item that T reads have committed.
  - *Cascadeless* schedule (to avoid cascading rollback): if every transaction in the schedule ***reads*** only items that were written by committed transactions.
  - *Strict* schedule: transactions can neither ***read*** nor ***write*** an item X until the last transaction that wrote X has committed (or aborted).
- *Nonrecoverable* schedules
  - *Nonrecoverable* schedule: once a transaction T is ***committed***, T is asked to be ***rolled-back***.

5.3. Characterizing transaction schedules based on recoverability

S_a': $r_1(X); r_2(X); w_1(X); r_1(Y); w_2(X); c_2; w_1(Y); c_1;$

S_a' is recoverable, even though it suffers from the *lost update problem*. → Why?

S_c: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); c_2; a_1;$

S_c is not recoverable. → Why?

S_d: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); c_1; c_2;$

S_d is recoverable. → Why?

S_e: $r_1(X); w_1(X); r_2(X); r_1(Y); w_2(X); w_1(Y); a_1; a_2;$

S_e is recoverable; but not casecadeless. → Why?

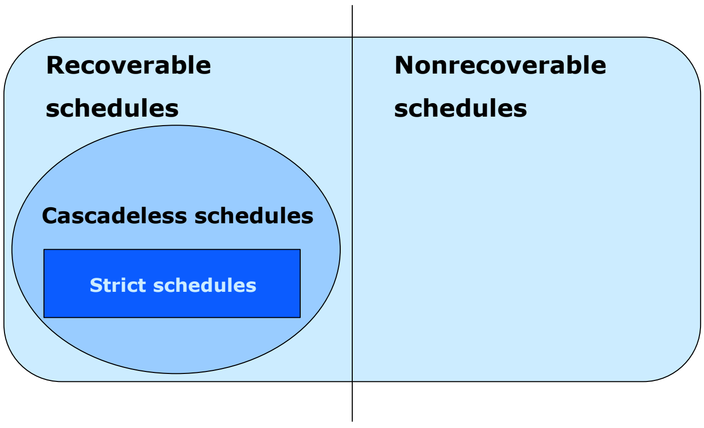
S_f: $r_1(X); w_1(X); r_1(Y); w_1(Y); c_1; r_2(X); w_2(X); c_2;$

S_f is casecadeless. → Why?

S_g: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); c_1; w_3(Y); c_3; r_2(Y); w_2(Z); r_2(Y); c_2;$

S_g is strict. → Why?

5.3. Characterizing transaction schedules based on recoverability



Relationships between schedules based on recoverability

All strict schedules are cascadeless.

All cascadeless schedules are recoverable.

33

5.3. Characterizing transaction schedules based on recoverability

- *Recoverable* schedules
  - *Recoverable* schedule
  - *Cascadeless* schedule (to avoid cascading rollback): if every transaction in the schedule *reads* only items that were written by committed transactions.
    - No log record for READ operations in the system log
  - *Strict* schedule: transactions can neither *read* nor *write* an item X until the last transaction that wrote X has committed (or aborted).
    - Log records for WRITE operations in the system log do not include new_value of data item X.
    - Strict schedules simplify the recovery process.
- *Nonrecoverable* schedules

34

5.4. Characterizing transaction schedules based on serializability

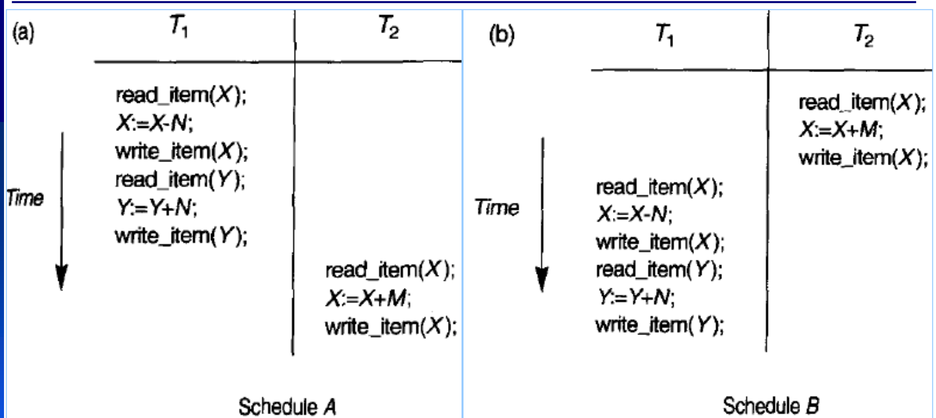


Figure 17.5. (a) Serial schedule A: T_1 followed by T_2. (b) Serial schedule B: T_2 followed by T_1.

Originally in DB: X = 5; Y = 10

Constants: N = 1; M = 2

Schedule A: X = 6; Y = 11

Schedule B: X = 6; Y = 11

5.4. Characterizing transaction schedules based on serializability

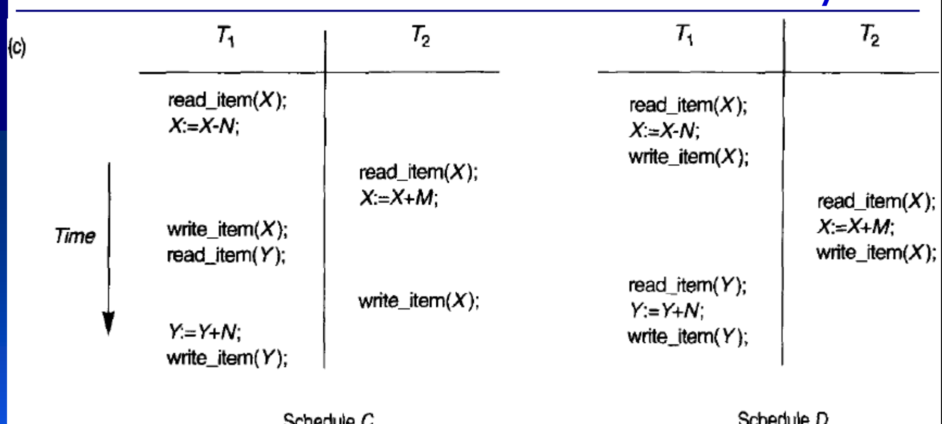


Figure 17.5. (c) Two nonserial schedules C and D with interleaving of operations.

Originally in DB: X = 5; Y = 10

Which schedules are correct with interleaving of operations? Why?

Constants: N = 1; M = 2

Schedule A: X = 6; Y = 11

Schedule B: X = 6; Y = 11

Schedule C: X = 7; Y = 11

Schedule D: X = 6; Y = 11

5.4. Characterizing transaction schedules based on serializability

- The concept of **serializability of schedules** is used to identify which schedules are **correct** when transaction executions have interleaving of their operations in the schedules.
- **Serial** schedules
  - The operations of each transaction are executed consecutively, without any interleaved operations from the other transaction.
- **Nonserial** schedules
  - The operations of each transaction are executed in an interleaved fashion with the operations of other transactions.

5.4. Characterizing transaction schedules based on serializability

- **Serial** schedules
  - Only one transaction at a time is active - the commit (or abort) of the active transaction initiates execution of the next transaction.
  - No interleaving occurs in a serial schedule.
  - If we consider the transactions to be *independent,* is that every serial schedule is considered correct.
  - The problem with serial schedules is that they limit concurrency or interleaving of operations.
    - if a transaction waits for an operation to complete, we cannot switch the CPU processor to another transaction, thus wasting valuable CPU processing time.
    - If some transaction T is quite long, the other transactions must wait for T to complete all its operations before commencing.

5.4. Characterizing transaction schedules based on serializability

- Nonserial schedules
  - Some nonserial schedules give the ***correct*** expected result.
  - We would like to determine which of the nonserial schedules always give a ***correct*** result and which may give ***erroneous*** results.
  - The concept used to characterize schedules in this manner is that of ***serializability*** of a schedule.

  → ***Serializable schedules***

5.4. Characterizing transaction schedules based on serializability

- Serializable schedules
  - A schedule S of *n* transactions is **serializable** if it is ***equivalent*** *to some serial schedule* of the same n transactions.
  - Two disjoint groups of the nonserial schedules:
    - Those that are equivalent to **one** (or **more**) of the serial schedules → **serializable** schedules
    - Those that are not equivalent to ***any*** serial schedule and hence are not serializable
  - Saying that a nonserial schedule S is serializable is equivalent to saying that it is **correct**, because it is equivalent to a serial schedule, which is considered correct.

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
    - Two schedules are called ***result equivalent*** if they produce the ***same*** final state of the database.
      - Two different schedules may accidentally produce the same final state.
      - Result equivalence alone cannot be used to define equivalence of schedules.
      - → The safest and most general approach to defining schedule equivalence is not to make any assumption about the types of operations included in the transactions.
      - → For two schedules to be equivalent, the operations applied to each data item affected by the schedules should be applied to that item in both schedules in *the same order*.
  - Conflict equivalence
  - View equivalence

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
    - Two schedules are said to be conflict equivalent if the order of any two *conflicting operations* is the same in both schedules.
      - Two schedules are said to be conflict equivalent if the order of any two *conflicting operations* is the same in both schedules.
        - Two operations in a schedule are said to *conflict* if they belong to different transactions, access the same database item, and at least one of the two operations is a write_item operation.

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence

Case 1:

S_a: … $w_1(X)$ … $r_2(X)$ … ≈ S_b: … $w_1(X)$ … $r_2(X)$ …

S_a: … $w_1(X)$ … $r_2(X)$ … ≠ $S_{b'}$: … $r_2(X)$ … $w_1(X)$ …

Case 2:

S_a: … $r_1(X)$ … $w_2(X)$ … ≈ S_b: … $r_1(X)$ … $w_2(X)$ …

S_a: … $r_1(X)$ … $w_2(X)$ … ≠ $S_{b'}$: … $w_2(X)$ … $r_1(X)$ …

Case 3:

S_a: … $w_1(X)$ … $w_2(X)$ … ≈ S_b: … $w_1(X)$ … $w_2(X)$ …

S_a: … $w_1(X)$ … $w_2(X)$ … ≠ $S_{b'}$: … $w_2(X)$ … $w_1(X)$ …

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
    - *Conflict serializable* schedule S if it is *conflict equivalent* to some <u>serial</u> schedule S'.
      - In such a case, we can reorder the *nonconflicting* operations in S until we form the equivalent serial schedule S'.
      - Schedules in Figure 17.5

S_a: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$ → serial?

S_b: $r_2(X)$; w2(X); $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$ → serial?

S_c: $r_1(X)$; r2(X); $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$ → not serializable?

S_d: $r_1(X)$; w1(X); $r_2(X)$; $w_2(X)$; $r_1(Y)$; $w_1(Y)$ → conflict serializable?

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
    - ***Conflict serializable*** schedule S → **Test** for conflict serializability of a schedule
      - The algorithm looks at only the read_item and write_itern operations in a schedule to construct a ***precedence graph*** (or ***serialization graph***), which is a directed graph G *(N,* E) that consists of a set of nodes N = $\{T_1, T_2, \ldots, T_n\}$ and a set of directed edges E = $\{e_1, e_2, \ldots, e_m\}$.
      - There is one node in the graph for each transaction T_i in the schedule.
      - Each edge e_i in the graph is of the form ($T_j \to T_k$), $1 \le j \le n$, $1 \le k \le n$, where T_j is the *starting node* of e_i and T_k is the *ending node* of e_i. Such an edge is created if one of the operations in T_j appears in the schedule *before* some *conflicting operation* in T_k.

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
    - ***Conflict serializable*** schedule S → Test for conflict serializability of a schedule

Algorithm 17.1: Testing conflict serializability of a schedule S.

1. For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph.
2. For each case in S where T_j executes a `read_item(X)` after T_i executes a `write_item(X)`, create an edge ($T_i \to T_j$) in the precedence graph.
3. For each case in S where T_j executes a `write_item(X)` after T_i executes a `read_item(X)`, create an edge ($T_i \to T_j$) in the precedence graph.
4. For each case in S where T_j executes a `write_item(X)` after T_i executes a `write_item(X)`, create an edge ($T_i \to T_j$) in the precedence graph.
5. The schedule S is serializable if and only if the precedence graph has no cycles.

[1], pp. 570.

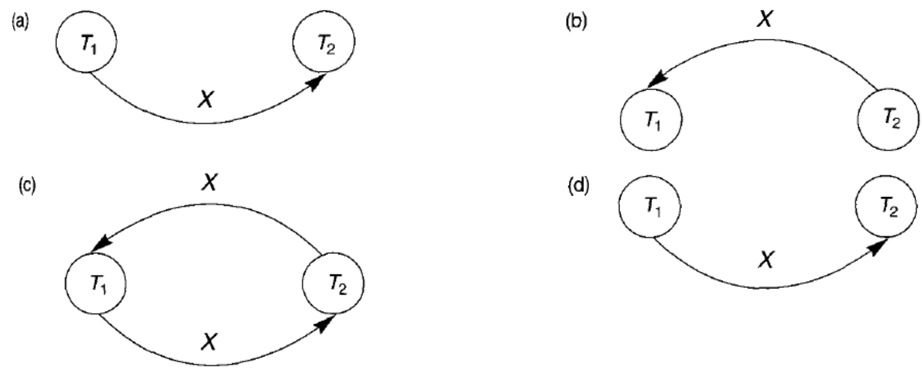


FIGURE 17.7 Constructing the precedence graphs for schedules *A* to *D* from Figure 17.5 to test for conflict serializability. (a) Precedence graph for serial schedule *A*. (b) Precedence graph for serial schedule *B*. (c) Precedence graph for schedule *C* (not serializable). (d) Precedence graph for schedule *D* (serializable, equivalent to schedule *A*).

S_a: $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$; $r_2(X)$; $w_2(X)$ → serial

S_b: $r_2(X)$; w2(X); $r_1(X)$; $w_1(X)$; $r_1(Y)$; $w_1(Y)$ → serial

S_c: $r_1(X)$; r2(X); $w_1(X)$; $r_1(Y)$; $w_2(X)$; $w_1(Y)$ → not serializable

S_d: $r_1(X)$; w1(X); $r_2(X)$; $w_2(X)$; $r_1(Y)$; $w_1(Y)$ → conflict serializable

5.4. Characterizing transaction schedules based on serializability

transaction T_1
read_item (*X*); write_item (*X*); read_item (*Y*); write_item (*Y*);

transaction T_2
read_item (*Z*); read_item (*Y*); write_item (*Y*); read_item (*X*); write_item (*X*);

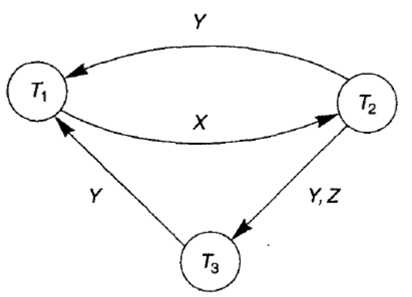
transaction T_3
read_item (*Y*); read_item (*Z*); write_item (*Y*); write_item (*Z*);

Figure 17.8, [1], pp. 572-574.

Another ***problem*** appears here: When transactions are submitted continuously to the system, it is *difficult to determine when a schedule begins and when it ends*. Serializability theory can be adapted to deal with this problem by considering only the committed projection of a schedule S. The *committed projection* C(S) of a schedule S includes only the operations in S that belong to committed transactions. We can theoretically define a schedule S to be serializable if its committed projection *C(S)* is equivalent to some serial schedule, since only committed transactions are guaranteed by the DBMS.

Time

transaction T_1	transaction T_2	transaction T_3
	read_item (Z); read_item (Y); write_item (Y);	
		read_item (Y); read_item (Z);
read_item (X); write_item (X);		
		write_item (Y); write_item (Z);
	read_item (X);	
read_item (Y); write_item (Y);		
	write_item (X);	



Equivalent serial schedules

None

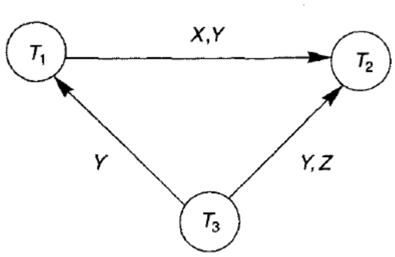
Reason

cycle $X(T_1 \rightarrow T_2)$, $Y(T_2 \rightarrow T_1)$
cycle $X(T_1 \rightarrow T_2)$, $YZ(T_2 \rightarrow T_3)$, $Y(T_3 \rightarrow T_1)$

49

Time

transaction T_1	transaction T_2	transaction T_3
		read_item (Y); read_item (Z);
read_item (X); write_item (X);		
		write_item (Y); write_item (Z);
	read_item (Z);	
read_item (Y); write_item (Y);		
	read_item (Y); write_item (Y); read_item (X); write_item (X);	



Equivalent serial schedules

$T_3 \longrightarrow T_1 \longrightarrow T_2$

50

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
  - View equivalence
    - Another *less restrictive* definition of equivalence of schedules
    - Two schedules S and S' are said to be ***view equivalent*** if the following three conditions hold:
      - 1. The same set of transactions participates in S and *S'*, and S and S' include the same operations of those transactions.
      - 2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in *S'*.
      - 3. If the operation $w_k(Y)$ of T_k is the last operation to write item *Y* in S, then $w_k(Y)$ *of* T_k must also be the last operation to write item *Y* in *S'*.

5.4. Characterizing transaction schedules based on serializability

- Equivalence of schedules
  - Result equivalence
  - Conflict equivalence
  - View equivalence
    - The *idea behind view equivalence* is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results.
    - The read operations are hence said to *see the same view* in both schedules.
    - Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules.
    - ***View serializable***: a schedule S is said to be ***view serializable*** if it is ***view equivalent*** to a <u>serial</u> schedule.

Concurrency control techniques

- Used to ensure the *noninterference* or *isolation property* of concurrently executing transactions
- Concurrency control techniques
  - The technique of locking data items to prevent multiple transactions from accessing the items concurrently (5.5)
    - Multiversion concurrency control (5.7)
  - Concurrency control protocols that use timestamps (5.6)
    - Multiversion concurrency control (5.7)
  - A protocol based on the concept of validation or certification of a transaction after it executes its operations (5.8)
  - Multiple granularity level two-phase locking protocol (5.9)

5.5. Two-phase locking techniques for concurrency control

- Based on the concept of *locking* data items to guarantee serializability of transaction schedules
- Two phases: expanding or growing (first) phase, shrinking (second) phase
  - A transaction is said to follow the *two-phase locking* protocol if *all locking* operations precede the *first unlock* operation in the transaction.
- Problems with locks: deadlock, starvation

Transaction T that follows the *two-phase locking* protocol

Expanding/growing (first) phase	Locking data items
Shrinking (second) phase	Unlocking data items

5.5. Two-phase locking techniques for concurrency control

- A ***lock*** is a variable associated with a data item that describes the status of the item with respect to possible operations that can be applied to it.
  - Used as a means of synchronizing the access by concurrent transactions to the database items
- Types of locks: binary locks, shared/exclusive (or read/write) locks, certify locks
  - System lock tables
  - Lock compatibility tables
  - Conversion of locks

55

5.5. Two-phase locking techniques for concurrency control

- Binary locks
  - Two states (values): locked (1) & unlocked (0)
  - If the value of the lock on data item X is **1**, i.e. LOCK(X), then item X *cannot be accessed* by a database operation (read/write) that requests the item.
  - If the value of the lock on X is **0**, then the item can be accessed when requested.
  - → A binary lock enforces *mutual exclusion* on the data item.
    - → At most one transaction can hold the lock on a data item.
- Two operations used with binary locking: lock_item, unlock_item

56

5.5. Two-phase locking techniques for concurrency control

```
lock_item (X):

  B: if LOCK (X)=0  (* item is unlocked *)
     then LOCK (X)←1  (* lock the item *)
     else begin
       wait (until lock (X)=0  and
         the lock manager wakes up the transaction);
       go to B
       end;

unlock_item (X):

  LOCK (X)←0;  (* unlock the item *)
  if any transactions are waiting
   then wakeup one of the waiting transactions;
```

Figure 18.1 Lock and unlock operations for binary locks.

[1], pp. 585.

5.5. Two-phase locking techniques for concurrency control

- Every transaction must obey the following rules in the *binary locking scheme*:
  - 1. A transaction T must issue the operation lock_item(X) before any read_item(X) or write_item(X) operations are performed in T.
  - 2. A transaction T must issue the operation unlock_item(X) after all read_item(X) and write_item(X) operations are completed in T.
  - 3. A transaction T will not issue a lock_item(X) operation if it already holds the lock on item X.
  - 4. A transaction will not issue an unlock_item(X) operation unless it already holds the lock on item X.

5.5. Two-phase locking techniques for concurrency control

- Shared/Exclusive (Read/Write) locks
  - Three states: read-locked, write-locked, unlocked
    - Read-locked: *share*-locked → other transactions are allowed to read the item.
    - Write-locked: *exclusive*-locked → a single transaction exclusively holds the lock on the item.
    - → *Less restrictive* than the binary locking scheme
- Three locking operations: **read_lock**(X), **write_lock**(X), **unlock**(X)

59

5.5. Two-phase locking techniques for concurrency control

```
read_lock (X):
 B: if LOCK (X)="unlocked"
     then begin LOCK (X)← "read-locked";
               no_of_reads(X)← 1
          end
     else if LOCK(X)="read-locked"
          then no_of_reads(X)← no_of _reads(X) + 1
          else begin wait (until LOCK (X)="unlocked" and
                     the lock manager wakes up the transaction);
                   go to B
               end;
```

Figure 18.2 Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.
[1], pp. 585.

60

5.5. Two-phase locking techniques for concurrency control

```
write_lock (X):
  B: if LOCK (X)="unlocked"
    then LOCK (X)← "write-locked"
    else begin
        wait (until LOCK(X)="unlocked" and
          the lock manager wakes up the transaction);
        go to B
        end;
```

Figure 18.2 Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.
[1], pp. 585.

5.5. Two-phase locking techniques for concurrency control

```
unlock (X):
  if LOCK (X)="write-locked"
    then begin LOCK (X)← "unlocked;"
        wakeup one of the waiting transactions, if any
        end
    else if LOCK(X)="read-locked"
        then begin
            no_of_reads(X)← no_of_reads(X) – 1;
            if no_of_reads(X)=0
              then begin LOCK (X)="unlocked";
                  wakeup one of the waiting transactions, if any
                  end
              end;
```

Figure 18.2 Locking and unlocking operations for two-mode (read-write or shared-exclusive) locks.
[1], pp. 585.

5.5. Two-phase locking techniques for concurrency control

- Every transaction must obey the following rules in the *shared/exclusive locking scheme*:
  - 1. A transaction T must issue the operation read_lock(X) or write_lock(X) before any read_item(X) operation is performed in T.
  - 2. A transaction T must issue the operation write_lock(X) before any write_item(X) operation is performed in T.
  - 3. A transaction T must issue the operation unlock(X) after all read_item(X) and write_item(X) operations are completed in T.
  - 4. A transaction T will not issue a read_lock(X) operation if it already holds a read (shared) lock or a write (exclusive) lock on item X.
  - 5. A transaction T will not issue a write_lock(X) operation if it already holds a read (shared) lock or write (exclusive) lock on item X.
  - 6. A transaction T will not issue an unlock(X) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item X.

Note: Rules 4 & 5 may be relaxed for ***lock conversion***.

63

5.5. Two-phase locking techniques for concurrency control

- Conversion of locks
  - A transaction that already holds a lock on item X is allowed under certain conditions to convert the lock from one locked state to another.
  - **Upgrade**: read_lock(X) → write_lock(X)
    - None of other transactions holds a lock on X.
  - **Downgrade**: write_lock(X) → read_lock(X)
  - When upgrading and downgrading of locks is used, the lock table must include *transaction identifiers* in the record structure for each lock to store the information on which transactions hold locks on the item.

64

5.5. Two-phase locking techniques for concurrency control

- Two-phase locking protocols
  - A transaction is said to follow the two-phase locking protocol if ***all locking*** operations (read_lock, write_lock) precede the ***first unlock*** operation in the transaction.
  - Such a transaction can be divided into two phases: an ***expanding or growing (first) phase***, during which new locks on items can be acquired but none can be released; a ***shrinking (second) phase***, during which existing locks can be released but no new locks can be acquired.
  - If lock conversion is allowed, then upgrading of locks must be done during the expanding phase, and downgrading of locks must be done in the shrinking phase.
  - If *every* transaction in a schedule follows the *two-phase locking* protocol, the schedule is ***guaranteed to be serializable***.
  - → When to lock data items?
  - → When to unlock data items?

5.5. Two-phase locking techniques for concurrency control

(a)

T_1	T_2
read_lock(Y);	read_lock(X);
read_item(Y);	read_item(X);
unlock(Y);	unlock(X);
write_lock(X);	write_lock(Y);
read_item(X);	read_item(Y);
X:=X+Y;	Y:=X+Y;
write_item(X);	write_item(Y);
unlock(X);	unlock(Y);

(b) Initial values: X=20, Y=30
Result of serial schedule T_1 followed by T_2 :
X=50, Y=80
Result of serial schedule T_2 followed by T_1 :
X=70, Y=50

Figure 18.3 Transactions that do *not* obey two-phase locking

[1], pp. 589.

5.5. Two-phase locking techniques for concurrency control

(c)

Result of schedule S:
X=50, Y=50
(nonserializable)

Time

T_1	T_2

read_lock(Y);
read_item(Y);
unlock(Y);	

	read_lock(X);
	read_item(X);
	unlock(X);
	write_lock(Y);
	read_item(Y);
	Y:=X+Y;
	write_item(Y);
	unlock(Y);

write_lock(X);
read_item(X);
X:=X+Y;
write_item(X);
unlock(X);	

Initial values: X=20, Y=30
Result of serial schedule T_1 followed by T_2 :
X=50, Y=80
Result of serial schedule T_2 followed by T_1 :
X=70, Y=50

5.5. Two-phase locking techniques for concurrency control

T_1'	T_2'	
read_lock (Y);	read_lock (X);	
read_item (Y);	read_item (X);	
write_lock (X);	write_lock (Y);	DEADLOCK
unlock (Y);	unlock (X);	
read_item (X);	read_item (Y);	
X:=X+Y;	Y:=X+Y;	
write_item (X);	write_item (Y);	
unlock (X);	unlock (Y);	

Figure 18.4 Transactions T_1' and T_2' which *follow* the two-phase locking protocol. Note that they can produce a *deadlock*.
[1], pp. 589.

5.5. Two-phase locking techniques for concurrency control

- Variations of two-phase locking (2PL)
  - Basic 2PL
  - Conservative 2PL (static 2PL)
    - A transaction locks all the items it accesses ***before*** the transaction begin execution, by predeclaring its read-set and write-set. If any of the predeclared items needed cannot be locked, the transaction does not lock any item. This is a deadlock-free protocol.
  - Strict 2PL
    - A transaction T does not release any of its ***exclusive (write)*** locks until ***after*** it commits or aborts. Hence, no other transaction can read or write an item that is written by T unless T has committed, leading to a strict schedule for recoverability. This is not deadlock-free.
  - Rigorous 2PL
    - A transaction T does not release any of its locks (***exclusive or shared***) until ***after*** it commits or aborts, leading to a strict schedule for recoverability but easier for implementation. This is not deadlock-free.

5.5. Two-phase locking techniques for concurrency control

- Deadlock
  - Deadlock occurs when *each* transaction T is a set of *two or more transactions* is waiting for some item that is locked by some other transaction T' in the set.
  - Deadlock ***prevention***
    - Transactions are long; each transaction uses many items; the transaction load is quite heavy.
  - Deadlock ***detection***
    - There will be little interference among the transactions – that is, different transactions will rarely access the same items at the same time; transactions are short; each transaction locks only a few items; the transaction load is light.

5.5. Two-phase locking techniques for concurrency control

- Deadlock prevention
  - Conservative 2PL
  - Timestamp-based deadlock prevention schemes: wait-die & wound-wait
    - Wait-die: if transaction T_i tries to lock an item X and is older than transaction T_j that currently locks X with a conflicting lock, T_i is allowed to ***wait***; otherwise T_i ***dies*** and is restarted *with the same timestamp*.
    - Wound-wait: If T_i is older than T_j, T_i ***wounds*** T_j, abort T_j and restart it later *with the same timestamp*; otherwise, T_i is allowed to ***wait***.
  - Deadlock prevention schemes that do not require timestamps: no-waiting & cautious-waiting
    - No-waiting: if a transaction is unable to obtain a lock, it is immediately aborted and then restarted after a certain time delay without checking whether a deadlock will actually occur or not.
    - Cautious-waiting: if T_j is not blocked (not waiting for some other locked item), then T_i is blocked and allowed to wait; otherwise abort T_i.

71

5.5. Two-phase locking techniques for concurrency control

- Deadlock detection
  - The system checks if a state of deadlock actually exists.
  - If the system is in a state of deadlock, some of the transactions causing the deadlock must be aborted.
  - Choosing which transactions to abort is known as ***victim selection***.
    - Avoid selecting transactions that have been running for a long time and that have performed many updates
  - A simple way to detect a state of deadlock is for the system to construct and maintain a ***wait-for graph***.

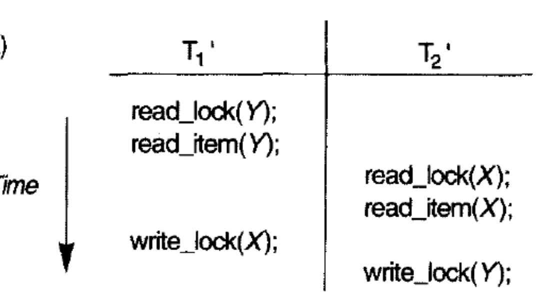
72

5.5. Two-phase locking techniques for concurrency control

- Deadlock detection
  - Construct and maintain a ***wait-for graph***
    - One node is created for each transaction that is currently executing.
    - A directed edge ($T_i \rightarrow T_j$) is created whenever a transaction T_i is waiting to lock and item X that is currently locked by a transaction T_j.
    - The directed edge ($T_i \rightarrow T_j$) is dropped when T_j releases the lock(s) on the items that T_i was waiting for.
    - → A state of deadlock *if and only if* the wait-for graph has a *cycle*.

5.5. Two-phase locking techniques for concurrency control

(a)



(b)

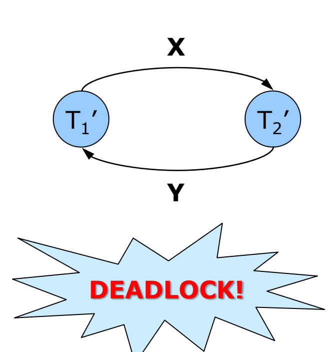


Figure 18.5 Illustrating the deadlock problem.

(a) A partial schedule of T_1' and T_2' that is in a state of deadlock.

(b) A *wait-for graph* for the partial schedule in (a).

[1], pp. 591.

5.5. Two-phase locking techniques for concurrency control

- Starvation
  - A transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
    - The waiting scheme for locked items is unfair, giving priority to some transactions over others.
    - Victim selection if the algorithm selects the same transaction as victim repeatedly, thus causing it to abort and never finish execution.
  - Solutions:
    - A fair waiting scheme
      - Using a first-come-first-served queue; transactions are enabled to lock an item in the order in which they originally requested the lock.
    - Priority-based schemes
    - Wait-die and wound-wait schemes

5.6. Timestamp-based techniques for concurrency control

- Guarantees serializability by using transaction timestamps to order transaction execution for an equivalent serial schedule
  - ***Timestamp ordering***: order the transactions based on their timestamps

  → Ensure: for each item accessed by *conflicting operations*, the order in which the item is accessed does not violate the ***serializability order***.

  → Deadlock-free

5.6. Timestamp-based techniques for concurrency control

- Transaction timestamp - **TS**(T): a unique identifier to identify a transaction T, assigned in the order in which the transaction is submitted to the system
- Read timestamp of item X – **read_TS**(X): the ***largest*** timestamp among all the timestamps of transactions that have successfully read item X – that is, read_TS(X) = TS(T), where T is the *youngest* transaction that has read X successfully
- Write timestamp of item X – **write_TS**(X): the ***largest*** of all the timestamps of transactions that have successfully written item X – that is, write_TS(X) = TS(T), where T is the *youngest* transaction that has written X successfully.

77

5.6. Timestamp-based techniques for concurrency control

- The basic timestamp ordering algorithm
  - Check if conflicting operations violate the timestamp ordering
  - → Schedules are guaranteed to be conflict serializability.
  - 1. Transaction T issues a write_item(X) operation
    - If read_TS(X) > TS(T) or if write_TS(X) > TS(T), then abort and roll back T and reject the operation because some younger transaction with a timestamp greater than TS(T) – and hence, after T in the timestamp ordering – has already read or written the value of item X before T had a chance to write X, thus violating the timestamp ordering.
    - Otherwise, execute the write_item(X) operation of T and set write_TS(X) to TS(T).
  - 2. Transaction T issues a read_item(X) operation
    - If write_TS(X) > TS(T), then abort and roll back T and reject the operation because some younger transaction with timestamp greater than TS(T) – and hence, after T in the timestamp ordering – has already written the value of item X before T had a chance to read X.
    - Otherwise, execute the read_item(X) operation of T and set read_TS(X) to the larger of TS(T) and the current read_TS(X).

78

5.6. Timestamp-based techniques for concurrency control

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)	Read_item(X)		
	Write_item(X)	Write_item(X)		
	Read_item(Y)			
	Write_item(Y)			

Serial schedule S of transactions T1 & T2 based on *timestamp*:

S = T1; T2 = r1(X); w1(X); r1(Y); w1(Y); r2(X); w2(X)

5.6. Timestamp-based techniques for concurrency control

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)		Read_TS = 1	
		Read_item(X)	Read_TS = 2	
	Write_item(X)		Reject!	
	Read_item(Y)			
		Write_item(X)	Write_TS = 2	
	Write_item(Y)			

Write_item(X) of T1 is *rejected*, T1 is *aborted*: Read_TS(X) > TS(T1)

Schedule C (Figure 17.5.c) is *non-serializable*.

5.6. Timestamp-based techniques for concurrency control

	T1	T2	X	Y
Timestamp	TS = 1	TS = 2	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
	Read_item(X)		Read_TS = 1	
	Write_item(X)		Write_TS = 1	
		Read_item(X)	Read_TS = 2	
		Write_item(X)	Write_TS = 2	
	Read_item(Y)			Read_TS = 1
	Write_item(Y)			Write_TS = 1

Schedule D (Figure 17.5.d) is *conflict serializable*.

5.6. Timestamp-based techniques for concurrency control

- Strict timestamp ordering
  - Ensure: the schedules are both *strict* (for easy recoverability) and *conflict serializable*
  - A transaction T that issues a *read_item(X)* or *write_item(X)* such that TS(T) > write_TS(X) has its read or write operation ***delayed*** until the transaction T' that wrote the value of X (hence, TS(T') = write_TS(X)) has committed or aborted.

5.6. Timestamp-based techniques for concurrency control

- Thomas's **Write** Rule
  - A modification of the basic timestamp ordering algorithm that does ***not*** enforce conflict serializability; but rejects fewer write operations
  - Checks for the **write_item(X)** operation:
    - 1. If read_TS(X) > TS(T), then abort and roll back T and reject the operation.
    - 2. If write_TS(X) > TS(T), then do not execute the write operation but continue processing. This is because some transaction with timestamp greater than TS(T) – and hence, after T in the timestamp ordering – has already written the value of X.
    - 3. Otherwise, execute the write_item(X) operation of T and set write_TS(X) to TS(T).

5.7. Multiversion concurrency control techniques

- Several ***versions*** of an item are maintained.
  - When a transaction writes an item, it writes a new version (new value) and the old version (old value) of the item is maintained.
- When a transaction requires access to an item, an ***appropriate version*** is chosen to maintain the serializability of the currently executing schedule.
- Some ***read*** operations can be accepted by reading an older version of the item to maintain serializability.

5.7. Multiversion concurrency control techniques

- Multiversion technique based on *timestamp ordering*
  - Several versions X_1, X_2, …, X_k of each data item X are maintained.
    - For each version, the value of version X_i and the two timestamps are kept:
      - Read_TS(X_i): the *largest* of all the timestamps of transactions that have successfully read version X_i
      - Write_TS(X_i): the timestamp of the transaction that wrote the value of version X_i
  - Whenever a transaction T is allowed to execute a write_item(X) operation, a new version X_{k+1} of item X is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to TS(T).

5.7. Multiversion concurrency control techniques

- Multiversion technique based on *timestamp ordering*
  - To ensure *serializability*, the following two rules are used:
    - 1. If transaction T issues a *write_item(X)* operation, and version **i** of X has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version X_j of X with read_TS(X_j) = write_TS(X_j) = TS(T).
    - 2. If transaction T issues a *read_item(X)* operation, find the version **i** of X that has the highest write_TS(X_i) of all versions of X that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

5.7. Multiversion concurrency control techniques

- Multiversion technique based on *timestamp ordering*

T1	T2	T3	X_0	Y_0	Z_0
TS = 20	TS = 25	TS = 15	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0	Read_TS = 0 Write_TS = 0
R1(X)	R2(Z)	R3(Y)			
W1(X)	R2(Y)	R3(Z)			
R1(Y)	W2(Y)	W3(Y)			
W1(Y)	R2(X)	W3(Z)			
	W2(X)				

Serial schedule S of transactions T1, T2, & T3 based on *timestamp*:

S = T3; T1; T2

5.7. Multiversion concurrency control techniques

T1	T2	T3	X_0	Y_0	Z_0	X_{20}	Y_{15}	Z_{15}
TS = 20	TS = 25	TS = 15	R=0 W=0	R=0 W=0	R=0 W=0	R=20 W=20	R=15 W=15	R=15 W=15
		R3(Y)		R=15				
		R3(Z)			R=15			
R1(X)			R=20					
W1(X)						created		
		W3(Y)					created	
		W3(Z)						created
	R2(Z)							R=25
R1(Y)							R=20	
W1(Y)								
	R2(Y)							
	W2(Y)							
	R2(X)							
	W2(X)							

What happens with the W1(Y) operation of transaction T1?

5.7. Multiversion concurrency control techniques

- Multiversion *two-phase locking* using certify locks
  - A multiple-mode locking scheme
    - Three locking modes for an item: read, write, certify
    - State of LOCK(X) for an item X: read-locked, write-locked, certify-locked, or unlocked

(a)

	Read	Write
Read	yes	no
Write	no	no

(b)

	Read	Write	Certify
Read	yes	yes	no
Write	yes	no	no
Certify	no	no	no

Figure 18.6 Lock compatibility tables.

(a) A compatibility table for read/write locking scheme

(b) A compatibility table for read/write/certify locking scheme

[1], pp. 598.

5.7. Multiversion concurrency control techniques

- Multiversion *two-phase locking* using certify locks
  - In the standard 2PL scheme, once a transaction obtains a write lock on an item X, no other transactions can access X.
  - In the multiversion 2PL scheme, other transactions T' are allowed to read an item X while a single transaction T holds a write lock on X.
    - Two versions for each item X: one version written by some committed transaction; another version X' created by transaction T that acquires a write lock on X
    - Other transactions (different from T) can continue to read the committed version of X while T holds the write lock.
    - Once T is ready to commit, T must obtain a certify lock on all items that it currently holds write locks on before it can commit.
    - Once the certify locks are acquired, the committed version X of the data item is set to the value of version X', version X' is discarded, and the certify locks are then released.
    - → No cascading aborts; but deadlocks may occur if upgrading of a read lock to a write lock is allowed.

5.7. Multiversion concurrency control techniques

- Multiversion ***two-phase locking*** using certify locks

T1’	T2’	X	Y
Read_lock(Y)	Read_lock(X)		
Read_item(Y)	Read_item(X)		
Write_lock(X)	Write_lock(Y)		
Unlock(Y)	Unlock(X)		
Read_item(X)	Read_item(Y)		
Write_item(X)	Write_item(Y)		
Unlock(X)	Unlock(Y)		

Figure 18.4 Transactions T1’ and T2’ that follow the 2PL protocol

- Multiversion ***two-phase locking*** using certify locks

T1’	T2’	$X_{committed}$	$Y_{committed}$	$X_{T1'}$
Read_lock(Y)			Read_locked (T1’)	
Read_item(Y)			T1’ reads	
Write_lock(X)		Write_locked (T1’)		
	Read_lock(X)	Read_locked (T2’)		
	Read_item(X)	T2’ reads		
Unlock(Y)			Unlocked (T1’)	
	Write_lock(Y)		Write_locked (T2’)	
Read_item(X)		T1’ reads		
Write_item(X)				created
	Unlock(X)	Unlocked (T2’)		
Certify_lock(X)		$X_{T1'} \rightarrow X_{committed}$		discarded
	Read_item(Y)		T2’ reads	
	Write_item(Y)		???	
	Certify_lock(Y)		???	

5.8. Validation (optimistic) concurrency control techniques

- No checking for concurrency control is done while the transaction is executing.
- Updates in the transaction are not applied directly to the database items until the transaction reaches its end.
- At the end of transaction execution, a validation phase checks whether any of the transaction's updates violate serializability.
- If serializability is not violated, the transaction is committed and the database is updated from the local copies; otherwise, the transaction is aborted and then restarted later.
- "Optimistic" because the techniques assume that little interference will occur and hence that there is no need to do checking during transaction execution.

5.8. Validation (optimistic) concurrency control techniques

- There are three phases for this concurrency control protocol:
  - 1. *Read phase*: a transaction can read values of committed data items from the database. Updates are applied only to local copies (versions) of the data items kept in the transaction workspace.
  - 2. *Validation phase*: checking is performed to ensure that serializability will not be violated if the transaction updates are applied to the database.
  - 3. *Write phase*: if the validation phase is successful, the transaction updates are applied to the database; otherwise, the updates are discarded and the transaction is restarted.

5.8. Validation (optimistic) concurrency control techniques

- In the ***validation phase*** for transaction T_i, the timestamp-based protocol checks that T_i does not interfere with any committed transactions or with any other transactions T_j currently in their validation phase.
  - 1. Transaction T_j completes its write phase before T_i starts its read phase.
  - 2. T_i starts its write phase after T_j completes its write phase, and the read_set of T_i has no items in common with the write_set of T_j.
  - 3. T_j completes its read phase before T_i completes its read phase; and both the read_set and write_set of T_i have no items in common with the write_set of T_j.
  - → Only if condition (1) is false, is condition (2) checked; only if (2) is false is (3) checked.
  - → If any one of these three conditions holds, there is no interference and T_i is validated successfully. If none, the validation of T_i fails and T_i is aborted and restarted later.

5.9. Multiple granularity level two-phase locking protocol for concurrency control

- In this ***multiple granularity*** locking scheme, the granularity level (size of the data item) may be changed dynamically.
  - A field value of a database record
  - A database record
  - A disk block
  - A whole file
  - The whole database
  - → *Fine granularity* refers to small item sizes; *coarse granularity* refers to large item sizes.
  - → The larger the data item size is, the lower the degree of concurrency permitted.
  - → The smaller the data item size is, the more the number of items in the database; leading to a larger number of active locks to be handled by the lock manager.

5.9. Multiple granularity level two-phase locking protocol for concurrency control

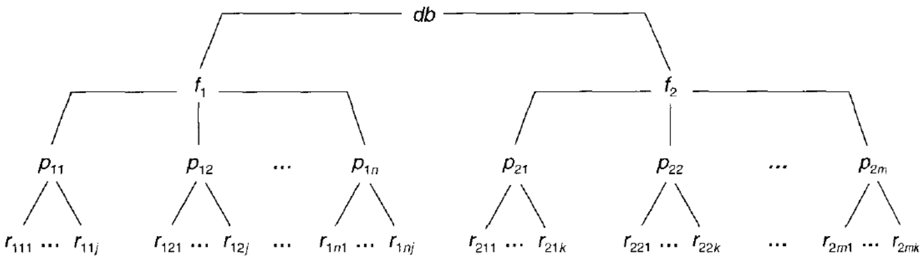


Figure 18.7 A granularity hierarchy for illustrating multiple granularity level locking. [1], pp. 602.

Database
↓
Files
↓
Pages
↓
Records

Where can a lock be requested?

5.9. Multiple granularity level two-phase locking protocol for concurrency control

- Types of locks
  - Shared lock (**S**) on node N: N is locked in shared mode.
  - Exclusive lock (**X**) on node N: N is locked in exclusive mode.
  - Intention-shared lock (**IS**) on node N: a shared lock(s) will be requested on some descendant node(s) of N.
  - Intention-exclusive (**IX**) on node N: an exclusive lock(s) will be requested on some descendant node(s) of N.
  - Shared-intention-exclusive (**SIX**) on node N: N is locked in shared mode but an exclusive lock(s) will be requested on some descendant node(s).
  - → *Intention locks*: what type of lock (shared or exclusive) a transaction will require from one of the node's descendants along the path from the root to the desired node.

5.9. Multiple granularity level two-phase locking protocol for concurrency control

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

Figure 18.8 Lock compatibility matrix for multiple granularity locking [1], pp. 603.

5.9. Multiple granularity level two-phase locking protocol for concurrency control

- The multiple granularity locking protocol consists of the following rules:
  - 1. The *lock compatibility* must be adhered to.
  - 2. The *root* of the tree must be locked first, in any mode.
  - 3. A node N can be locked by a transaction T in *S* or *IS* mode only if the parent of node N is already locked by transaction T in either IS or IX mode.
  - 4. A node N can be locked by a transaction T in *X*, *IX*, or *SIX* mode only if the parent of node N is already locked by transaction T in either IX or SIX mode.
  - 5. A transaction T can lock a node only if it has not unlocked any node (to enforce the 2PL protocol for serializability).
  - 6. A transaction T can unlock a node, N, only if none of the children of node N are currently locked by T.

T1	T2	T3
IX(db)		
IX(f_1)		
	IX(db)	
		IS(db)
		IS(f_1)
		IS(p_{11})
IX(p_{11})		
X(r_{111})		
	IX(f_1)	
	X(p_{12})	
		S(r_{11j})
IX(f_2)		
IX(p_{21})		
X(r_{211})		

(cont.)

Unlock(r_{211})		
Unlock(p_{21})		
Unlock(f_2)		
		S(f_2)
	Unlock(p_{12})	
	Unlock(f_1)	
	Unlock(db)	
Unlock(r_{111})		
Unlock(p_{11})		
Unlock(f_1)		
Unlock(db)		
		Unlock(r_{11j})
		Unlock(p_{11})
		Unlock(f_1)
		Unlock(f_2)
		Unlock(db)

Using the granularity hierarchy above, consider the following three transactions in a possible serializable schedule (only the lock operations are shown):

1. T1 wants to update record r_{111} and record r_{211}.
2. T2 wants to update all records on page p_{12}.
3. T3 wants to read record r_{11j} and the entire f_2 file.

5.10. Conclusion

- ***Transaction***: a logical unit of work (database processing: read/write) with ***ACID*** properties
- ***Schedule of transactions***: recoverability, serializability
- ***Execution environment***: single-user/multi-user
  - The need of ***concurrency control*** in a multi-user environment
    - Two-phase locking concurrency control
      - Basic, conservative, strict, multiversion with certify locks, multiple granularity level
    - Timestamp-based concurrency control
      - Basic, strict, multiversion
    - Validation (optimistic) concurrency control
  - The need of ***recovery*** & ***backup*** for transaction failures
    - Recovery log

Questions ???

Review

- Review questions and exercises in [1], chapters 17-18.

Next - Chapter 6: Database recovery

- 6.1. An overview of database recovery
- 6.2. Deferred update-based recovery techniques
- 6.3. Immediate update-based recovery techniques
- 6.4. Shadow paging
- 6.5. The ARIES recovery algorithm
- 6.6. Conclusion

→ *Reading*: chapter 19, [1], pp. 611-635.