

Databases 2022

Darko Bozhinoski,
Ph.D. in Computer Science

Agenda

- Introduction to Transaction Processing
- Issues with concurrent run of transactions
- Transaction Execution and Transaction States
- ACID Properties
- Schedules and their classification
- Schedule equivalence
- Transaction Support in SQL
- Database concurrency control: Two-Phase Locking Techniques

Introduction to Transaction Processing

- **Single-User System:**
 - At most one user at a time can use the system.
- **Multuser System:**
 - Many users can access the system concurrently.
- **Concurrency**
 - **Interleaved processing:**
 - Concurrent execution of processes is interleaved in a single CPU
 - **Parallel processing:**
 - Processes are concurrently executed in multiple CPUs.

Transaction Definition

- **A Transaction:**
 - Logical unit of database processing that includes one or more access operations (read -retrieval, write - insert or update, delete).
- A transaction (set of operations) may be stand-alone specified in a high level language like SQL submitted interactively, or may be embedded within a program.
- **Transaction boundaries:**
 - Begin and End transaction.
- An **application program** may contain several transactions separated by the Begin and End transaction boundaries.

Transaction Basic Operations

SIMPLE MODEL OF A DATABASE (for purposes of discussing transactions):

- **A database** is a collection of named data items
- **Granularity** of data - a field, a record , or a whole disk block (Concepts are independent of granularity)
- Basic operations are **read** and **write**
 - **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.

READ AND WRITE OPERATIONS

- Basic unit of data transfer from the disk to the computer main memory is one block. In general, a data item (what is read or written) will be the field of some record in the database, although it may be a larger unit such as a record or even a whole block.
- `read_item(X)` command includes the following steps:
 - Find the address of the disk block that contains item X.
 - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
 - Copy item X from the buffer to the program variable named X.

READ AND WRITE OPERATIONS (2)

READ AND WRITE OPERATIONS:

- **write_item(X)** command includes the following steps:
 - Find the address of the disk block that contains item X.
 - Copy that disk block into a buffer in main memory (if that disk block is not already in some main memory buffer).
 - Copy item X from the program variable named X into its correct location in the buffer.
 - Store the updated block from the buffer back to disk (either immediately or at some later point in time).

Two sample transactions

- Two sample transactions:
 - (a) Transaction T1
 - (b) Transaction T2

(a) T_1

read_item (X);
 $X := X - N$;
write_item (X);
read_item (Y);
 $Y := Y + N$;
write_item (Y);

(b) T_2

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

Two sample transactions

- Two sample transactions:
 - (a) Transaction T1
 - (b) Transaction T2

What problems might happen if T1 and T2 are executed in parallel?

(a) T_1

read_item (X);
 $X := X - N$;
write_item (X);
read_item (Y);
 $Y := Y + N$;
write_item (Y);

(b) T_2

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

Why Concurrency Control is needed?

- **The Lost Update Problem**

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

- **The Temporary Update (or Dirty Read) Problem**

- This occurs when one transaction updates a database item and then the transaction fails for some reason (see Section 17.1.4).
- The updated item is accessed by another transaction before it is changed back to its original value.

- **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.

Why Concurrency Control is needed? (2)

- **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.

- **The Unrepeatable Read Problem.**

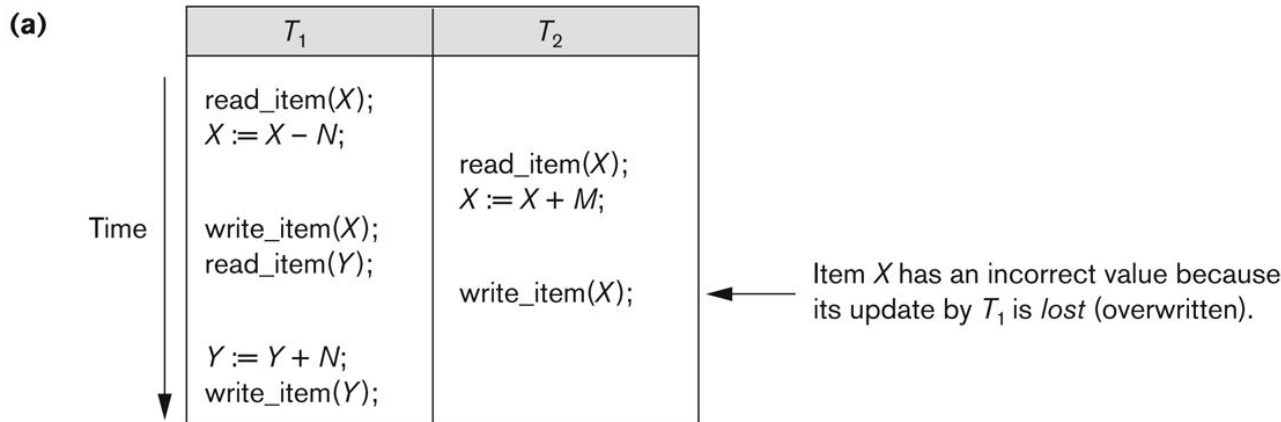
- Another problem that may occur is called *unrepeatable read*, where a transaction T reads the same item twice and the item is changed by another transaction T' between the two reads.

Concurrent execution is uncontrolled:

(a) The lost update problem.

Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



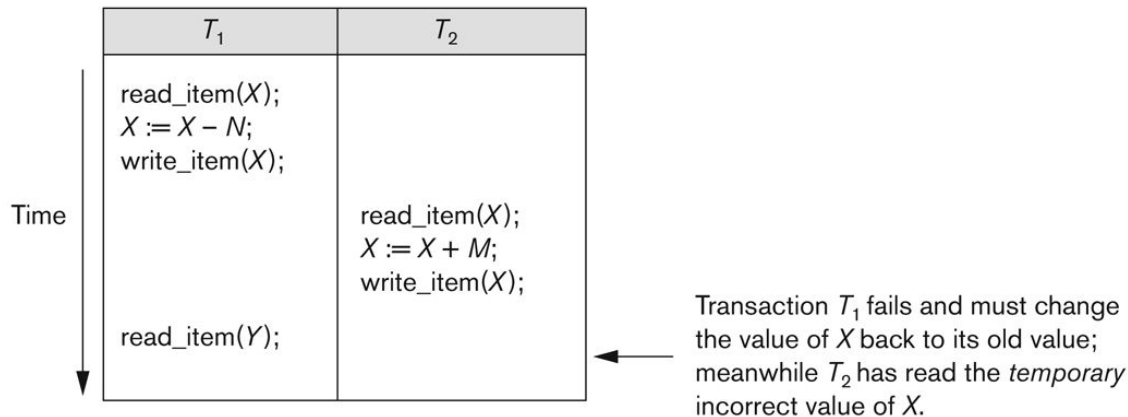
Concurrent execution is uncontrolled:

(b) The temporary update problem.

Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(b)



Concurrent execution is uncontrolled:

(c) The incorrect summary problem.

Figure 17.3

Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

(c)

T_1	T_3
<pre>read_item(X); X := X - N; write_item(X); read_item(Y); Y := Y + N; write_item(Y);</pre>	<pre>sum := 0; read_item(A); sum := sum + A; ⋮ read_item(X); sum := sum + X; read_item(Y); sum := sum + Y;</pre>

← T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

What causes a transaction to fail?

Why **recovery** is needed:

1. **A computer failure (system crash):**

A hardware or software error occurs in the computer system during transaction execution. If the hardware crashes, the contents of the computer's internal memory may be lost.

2. **A transaction or system error:**

Some operation in the transaction may cause it to fail, such as integer overflow or division by zero. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, the user may interrupt the transaction during its execution.

What causes a transaction to fail? (2)

3. Local errors or exception conditions detected by the transaction:

Certain conditions necessitate cancellation of the transaction. For example, data for the transaction may not be found. A condition, such as insufficient account balance in a banking database, may cause a transaction, such as a fund withdrawal from that account, to be canceled.

A programmed abort in the transaction causes it to fail.

4. Concurrency control enforcement:

The concurrency control method may decide to abort the transaction, to be restarted later, because it violates serializability or because several transactions are in a state of deadlock.

What causes a transaction to fail? (3)

5. Disk failure:

Some disk blocks may lose their data because of a read or write malfunction or because of a disk read/write head crash. This may happen during a read or a write operation of the transaction.

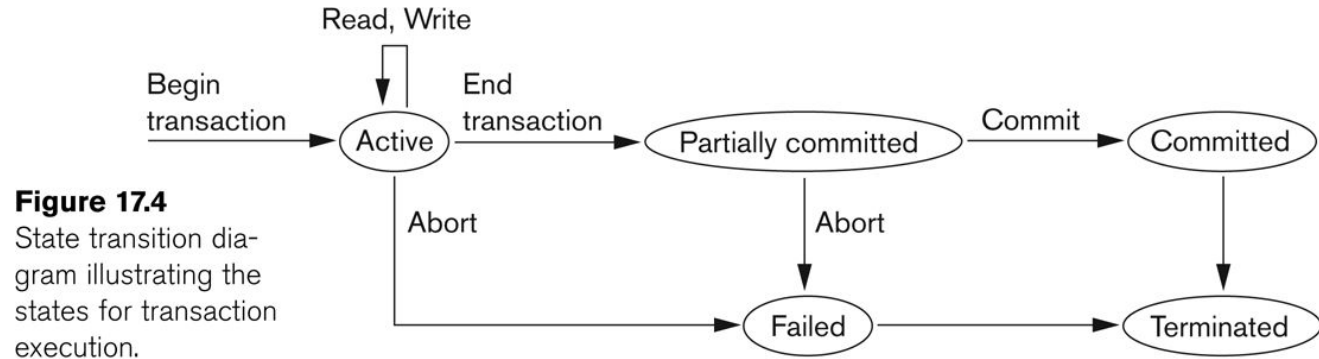
6. Physical problems and catastrophes:

This refers to an endless list of problems that includes power or air-conditioning failure, fire, theft, sabotage, overwriting disks or tapes by mistake, and mounting of a wrong tape by the operator.

Transaction Execution

- A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.
 - For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- **Transaction states:**
 - Active state
 - Partially committed state
 - Committed state
 - Failed state
 - Terminated State

Lifecycle of transaction execution



Transaction Execution: Recovery Manager

- **Recovery manager** keeps track of the following operations:
 - **begin_transaction**: This marks the beginning of transaction execution.
 - **read** or **write**: These specify read or write operations on the database items that are executed as part of a transaction.

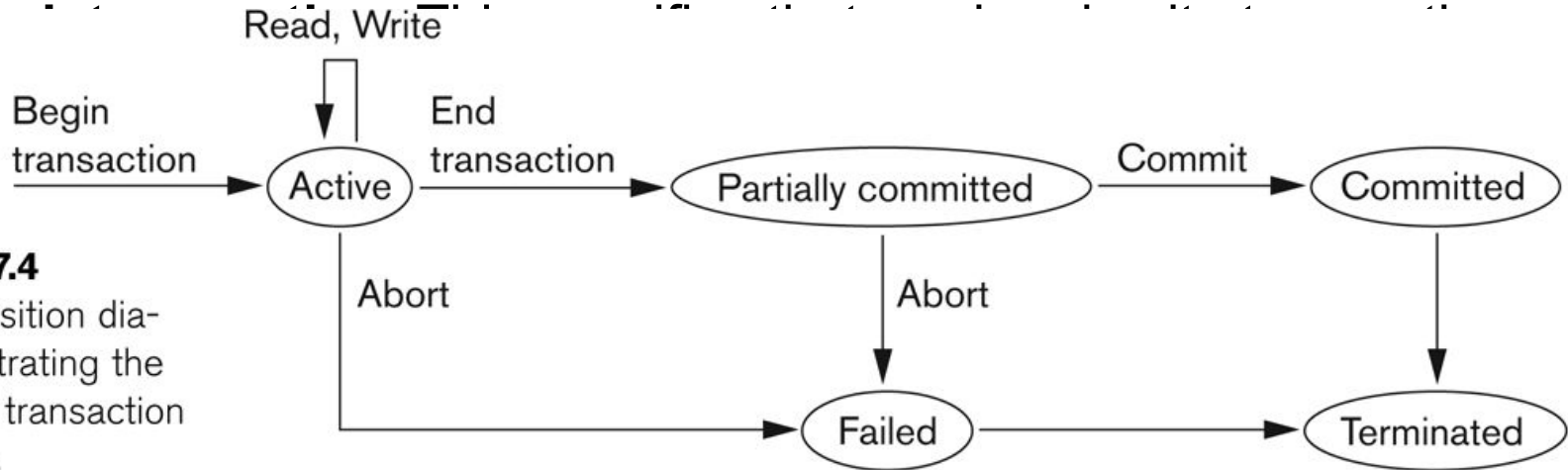


Figure 17.4

State transition diagram illustrating the states for transaction execution.

Transaction Execution: Recovery Manager (2)

- **end_transaction:** This specifies that read and write transaction operations have ended and marks the end limit of transaction execution.
 - At this point it may be necessary to check whether the changes introduced by the transaction can be permanently applied to the database or whether the transaction has to be aborted because it violates concurrency control or for some other reason.

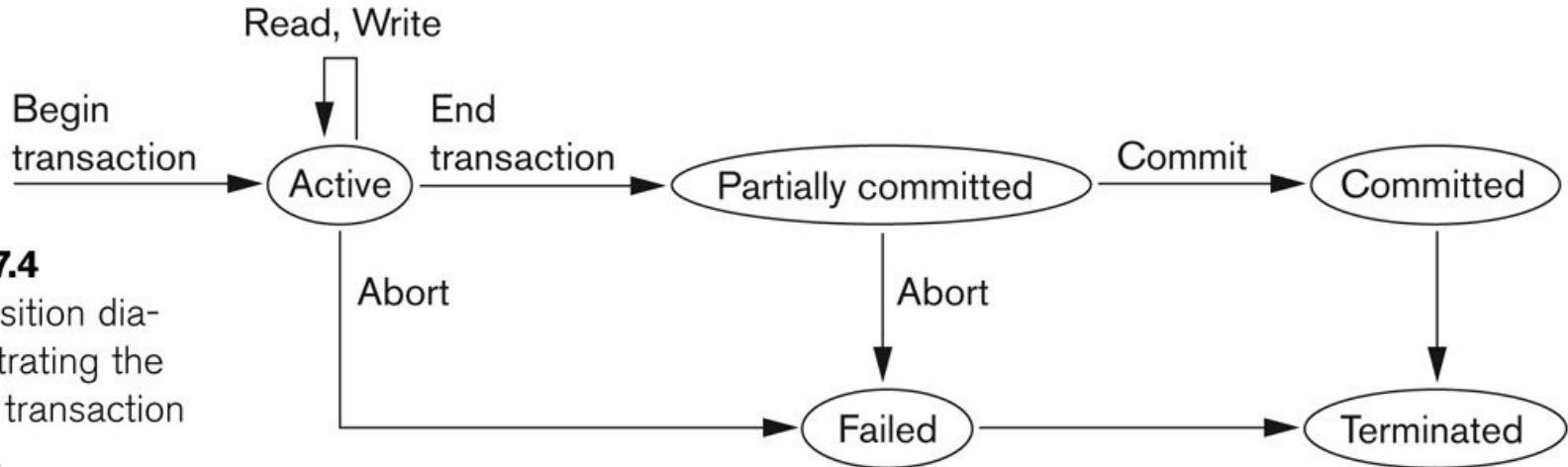


Figure 17.4

State transition diagram illustrating the states for transaction execution.

Transaction Execution: Recovery Manager (3)

- **commit_transaction**: This signals a successful end of the transaction so that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- **rollback** (or **abort**): This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

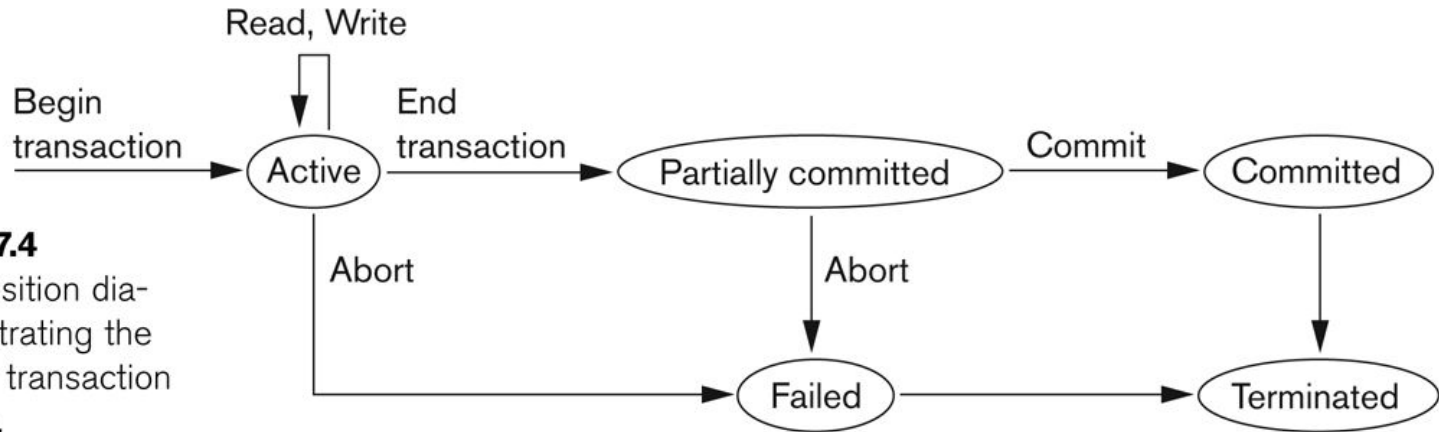


Figure 17.4

State transition diagram illustrating the states for transaction execution.

Recovery techniques operators

- Recovery techniques use the following operators:
 - **undo**: Similar to rollback except that it applies to a single operation rather than to a whole transaction.
 - **redo**: This specifies that certain *transaction operations* must be *redone* to ensure that all the operations of a committed transaction have been applied successfully to the database.

System Log

- **Log or Journal:** The log keeps track of all transaction operations that affect the values of database items.
 - This information may be needed to permit recovery from transaction failures.
 - The log is a sequential, append-only file kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure.
 - In addition, the log is periodically backed up to archival storage (tape) to guard against such catastrophic failures.

System Log (2)

- **Types of log record:**

- [start_transaction,T]: Records that transaction T has started execution.
- [write_item,T,X,old_value,new_value]: Records that transaction T has changed the value of database item X from old_value to new_value.
- [read_item,T,X]: Records that transaction T has read the value of database item X.
- [commit,T]: Records that transaction T has completed successfully, and affirms that its effect can be committed (recorded permanently) to the database.
- [abort,T]: Records that transaction T has been aborted.

Note: T refers to a unique **transaction-id** that is generated automatically by the system and is used to identify each transaction.

System Log (3)

- Protocols for recovery that *avoid cascading rollbacks do not require that read operations be written to the system log*, whereas other protocols require these entries for recovery.
- if the log is also used for other purposes—such as auditing (keeping track of all database operations)—then such entries can be included.
- Some recovery protocols require simpler WRITE entries that only include one of `new_value` or `old_value` instead of including both

Recovery using log records

- If the system crashes, we can recover to a consistent database state by examining the log and using some recovery techniques.
 1. Because the log contains a record of every write operation that changes the value of some database item, it is possible to **undo** 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.
 2. We can also **redo** the effect of the write operations of a transaction T by tracing forward through the log and setting all items changed by a write operation of T (that did not get done permanently) to their new_values.

Commit Point of a Transaction

- **Definition of a Commit Point:**

- 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 has 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.
- The transaction then writes an entry [**commit**, T] into the log.

- **Roll Back of transactions:**

- Needed for transactions that have a [start_transaction, T] entry into the log but no commit entry [**commit**, T] into the log.

Commit Point of a Transaction (2)

- **Redoing transactions:**

- Transactions that have written their commit entry in the log must also have recorded all their write operations in the log; otherwise they would not be committed, so their effect on the database can be redone from the log entries. (NOTE: the log file must be kept on disk.)
- At the time of a system crash, only the **log entries that have been written back to disk are considered in the recovery process** because the contents of main memory may be lost.

- **Force writing a log:**

- Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the disk.
- This process is called **force-writing the log** file before committing a transaction.

ACID Properties

- **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation:** A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation:** A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary.
- **Durability or permanency:** Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

ACID Properties

- **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation:** A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation:** A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly,

solves the temporary update problem and makes succeeding rollbacks safe.

The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

Schedule

- **Transaction schedule or history:**
 - When transactions are executing concurrently in an interleaved fashion, the order of execution of operations from the various transactions forms what is known as a transaction schedule (or history).
- A **schedule (or history)** S of n transactions T_1, T_2, \dots, T_n :
 - It is an ordering of the operations of the transactions subject to the constraint that, 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 .
 - Note, however, that operations from other transactions T_j can be interleaved with the operations of T_i in S .

Conflict operations in a schedule

Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:

- ❖ they belong to different transactions;
- ❖ they access the same item X;
- ❖ at least one of the operations is a write_item(X).

Intuitively, two operations are conflicting if changing their order can result in a different outcome.

Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y); - Which operations are conflicting?

Conflict operations in a schedule

Two operations in a schedule are said to conflict if they satisfy all three of the following conditions:

- ❖ they belong to different transactions;
- ❖ they access the same item X;
- ❖ at least one of the operations is a write_item(X).

Intuitively, two operations are conflicting if changing their order can result in a different outcome.

Sa: r1(X); r2(X); w1(X); r1(Y); w2(X); w1(Y); - Which operations are conflicting?

r1(X) and w2(X); r2(X) and w1(X); w1(X) and w2(X).

Complete schedule

A schedule S of n transactions T_1, T_2, \dots, T_n is said to be a complete schedule if the following conditions hold:

- The operations in S are exactly those operations in T_1, T_2, \dots, T_n , including a commit or abort operation as the last operation for each transaction in the schedule.
- For any pair of operations from the same transaction T_i , their relative order of appearance in S is the same as their order of appearance in T_i .
- For any two conflicting operations, one of the two must occur before the other in the schedule.

Complete schedule (2)

A schedule S of n transactions T_1, T_2, \dots, T_n is said to be a complete schedule if the following conditions hold:

- The operations in S are exactly those operations in T_1, T_2, \dots, T_n , including a commit or abort operation as the last operation for each transaction in the schedule.
- For any pair of operations from the same transaction T_i , their relative order of appearance in S is the same as their order of appearance in T_i .
- For any two conflicting operations, one of the two must occur before the other in the schedule.

A complete schedule will not contain any active transactions at the end of the schedule!

Committed projection

It is difficult to encounter complete schedules in a transaction processing system because new transactions are continually being submitted to the system.

- ❖ **Committed projection $C(S)$ of a schedule S** includes only the operations in S that belong to committed transactions—that is, transactions whose commit operation is in S .

Schedules classified on Recoverability

- **Recoverable schedule:**

- One where no transaction that is committed needs to be rolled back. (durability property of transactions is not violated)
- A schedule S is recoverable if no transaction T in S commits until all transactions T' that have written an item that T reads have committed.

- **Cascadeless schedule:**

- One where every transaction reads only the items that are written by committed transactions.

Examples

Which of these schedules are recoverable?

Sa' : $r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;$

Sc : $r1(X); w1(X); r2(X); r1(Y); w2(X); c2; a1;$

Sd : $r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;$

Se : $r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;$

Examples

Which of these schedules are recoverable?

Sa': $r1(X); r2(X); w1(X); r1(Y); w2(X); c2; w1(Y); c1;$ - Recoverable.

Sc: $r1(X); w1(X); \underline{r2(X)}; r1(Y); w2(X); \underline{c2}; \underline{a1};$ - Not recoverable because T2 reads item X from T1, but T2 commits before T1 commits. Problem appears when a1 aborts after c2 commits.

Sd: $r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); c1; c2;$ - Recoverable. Difference is that c2 operation is postponed until after T1 commits.

Se: $r1(X); w1(X); r2(X); r1(Y); w2(X); w1(Y); a1; a2;$ - Recoverable. Cascading rollback, an uncommitted transaction has to be rolled back because it read an item from a transaction that failed.

Schedules classified on Recoverability (2)

- **Schedules requiring cascaded rollback:**
 - A schedule in which uncommitted transactions that read an item from a failed transaction must be rolled back.
- **Strict Schedules:**
 - A schedule in which a transaction can neither read or write an item X until the last transaction that wrote X has committed.

Schedules based on Serializability

- **Serial schedule:**

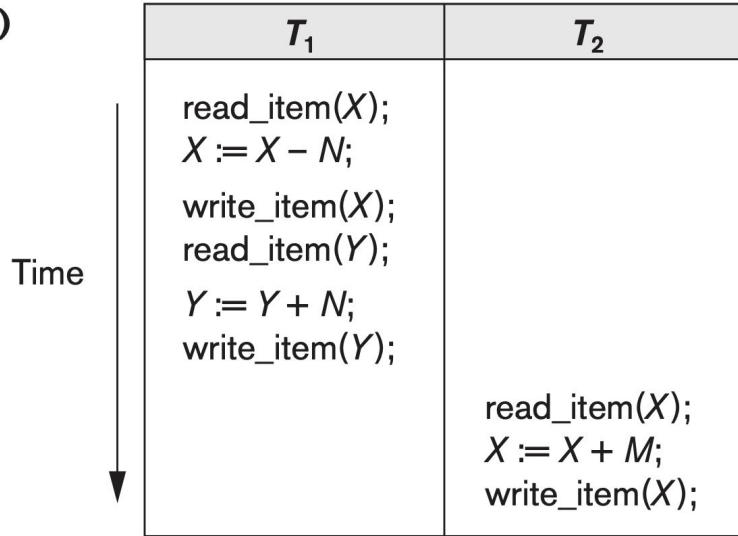
- A schedule S is serial if, for every transaction T participating in the schedule, all the operations of T are executed consecutively in the schedule.
 - Otherwise, the schedule is called nonserial schedule.

- **Every serial schedule is considered correct.**

- Serial schedules are unacceptable in practice...

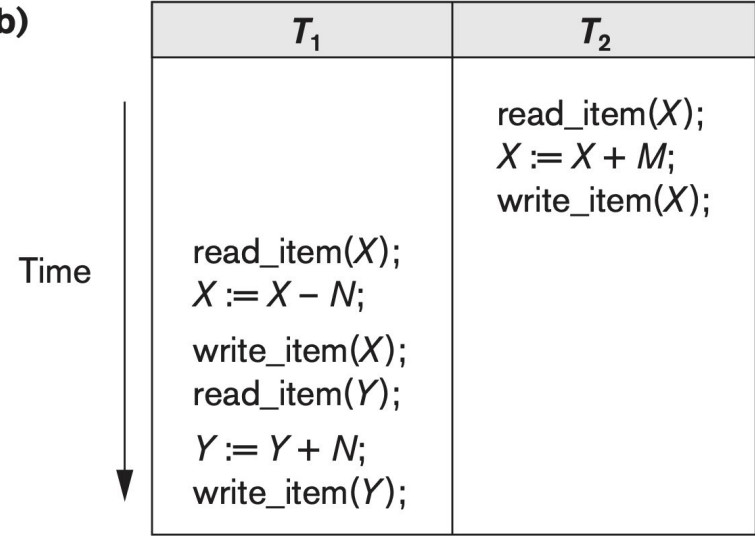
Examples of serial schedule

(a)



Schedule A

(b)



Schedule B

Serializable schedule

- **Serializable schedule:**

- A schedule S is serializable if it is equivalent to some serial schedule of the same n transactions.
- Resolves the issue with serial schedule: they limit concurrency by prohibiting interleaving of operations.

When are two schedules considered equivalent?

- **Result equivalent:**

- Two schedules are called result equivalent if they produce the same final state of the database.

- **Conflict equivalent:**

- Two schedules are said to be conflict equivalent if the order of any two conflicting operations is the same in both schedules.

- **Conflict serializable:**

- A schedule S is said to be conflict serializable if it is conflict equivalent to some serial schedule S' .

Example

Figure 20.6

Two schedules that are result equivalent for the initial value of $X = 100$ but are not result equivalent in general.

S_1
read_item(X); $X := X + 10$; write_item(X);

S_2
read_item(X); $X := X * 1.1$; write_item (X);

Result equivalent schedules for initial value $X=100$

Schedules based on Serializability (3)

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is a correct schedule.
 - It will leave the database in a consistent state.
 - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

Testing for Serializability of a Schedule

- Serializability is hard to check.
 - Interleaving of operations occurs in an operating system through some scheduler
 - Difficult to determine beforehand how the operations in a schedule will be interleaved.
- Most concurrency control methods do not actually test for serializability.

Practical approach to creating serializable schedules

- Protocols, or rules, are developed that guarantee that any schedule that follows these rules will be serializable.
- It's not possible to determine when a schedule begins and when it ends.
 - We reduce the problem of checking the whole schedule to checking only a **committed projection** of the schedule (i.e. operations from only the committed transactions.)
- Current approach used in most DBMSs:
 - Use of locks with **two phase locking**

View equivalence and View serializability

- **View equivalence:**
 - A less restrictive definition of equivalence of schedules
- **View serializability:**
 - Definition of serializability based on view equivalence.
 - A schedule is *view serializable* if it is *view equivalent* to a serial schedule.

View equivalence

- Two schedules 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' .

View equivalence (2)

- The premise behind view equivalence:
 - 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 view”**: the read operations are said to see *the same view* in both schedules.

Relationship between view and conflict equivalence

- The two are same under **constrained write assumption** which assumes that if T writes X , it is constrained by the value of X it read; i.e., $\text{new } X = f(\text{old } X)$
- Conflict serializability is **stricter** than view serializability. With unconstrained write (or blind write), a schedule that is view serializable is not necessarily conflict serializable.
 - Blind writes a write operation in a transaction T on an item X that is not dependent on the old value of X , so it is not preceded by a read of X in the transaction T .
 - The definitions of conflict serializability and view serializability are similar if a condition known as the **constrained write assumption** (or **no blind writes**) holds on all transactions in the schedule
- Any conflict serializable schedule is also view serializable, but not vice versa.

Relationship between view and conflict equivalence (2)

- Consider the following schedule of three transactions
 - T1: $r_1(X)$, $w_1(X)$; T2: $w_2(X)$; and T3: $w_3(X)$:
- Schedule Sa: $r_1(X)$; $w_2(X)$; $w_1(X)$; $w_3(X)$; c1; c2; c3;
- In Sa, the operations $w_2(X)$ and $w_3(X)$ are blind writes, since T1 and T3 do not read the value of X.
 - Sa is view serializable, since it is view equivalent to the serial schedule T1, T2, T3.
 - However, Sa is not conflict serializable, since it is not conflict equivalent to any serial schedule.

Algorithm for testing for conflict serializability

- Looks at only read_Item (X) and write_Item (X) operations
- Constructs a precedence graph (serialization graph) - a graph with directed edges
- An edge is created from T_i to T_j if one of the operations in T_i appears before a conflicting operation in T_j
- The schedule is serializable if and only if the precedence graph has no cycles.

Construct the Graph for these schedules.

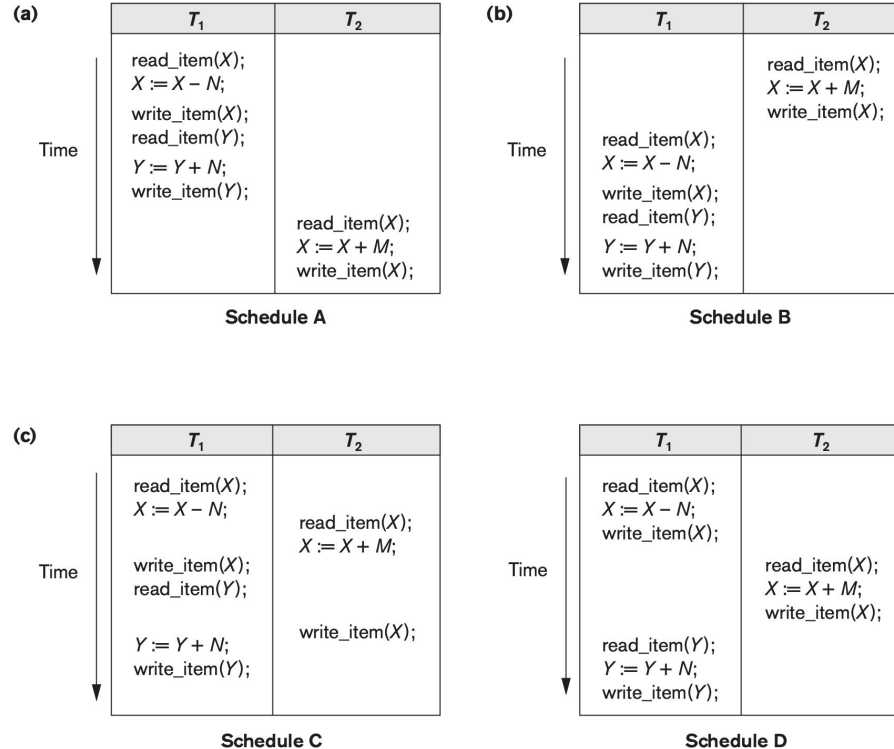
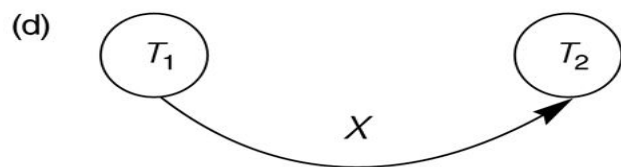
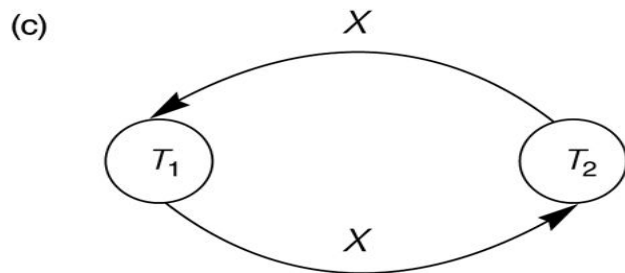
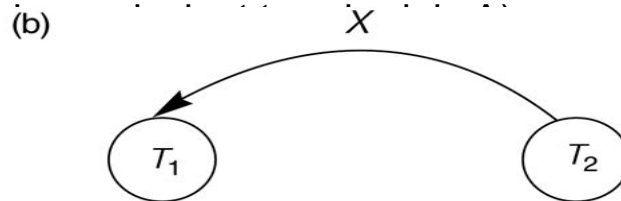
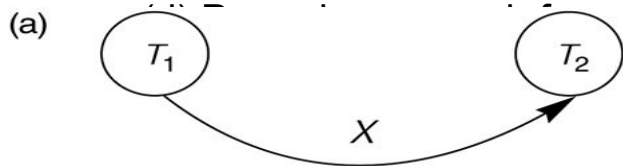


Figure 20.5

Examples of serial and nonserial schedules involving transactions T_1 and T_2 . (a) Serial schedule A: T_1 followed by T_2 . (b) Serial schedule B: T_2 followed by T_1 . (c) Two nonserial schedules C and D with interleaving of operations.

Constructing the Precedence Graphs

- Constructing the precedence graphs for schedules A and D 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).



Other Types of Equivalence of Schedules

- Under special **semantic constraints**, schedules that are otherwise not conflict serializable may work correctly (e.g. as debit-credit transactions)
 - Using commutative operations of addition and subtraction (which can be done in any order) certain non-serializable transactions may work correctly

Transaction Support in SQL

- A **single** SQL statement is always considered to be **atomic**.
 - Either the statement completes execution without error or it fails and leaves the database unchanged.
- With SQL, there is no explicit Begin Transaction statement.
 - Transaction initiation is done implicitly when particular SQL statements are encountered.
- Every transaction must have an explicit end statement, which is either a COMMIT or ROLLBACK.

SET TRANSACTION statement in SQL

- **Access mode:**
 - READ ONLY or READ WRITE.
 - The default is READ WRITE unless the isolation level of READ UNCOMMITTED is specified, in which case READ ONLY is assumed.
- **Diagnostic size** n, specifies an integer value n, indicating the number of conditions that can be held simultaneously in the diagnostic area.

SET TRANSACTION statement in SQL (2)

- **Isolation level** <isolation>, where <isolation> can be READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ or SERIALIZABLE. The default is SERIALIZABLE.
 - With SERIALIZABLE: the interleaved execution of transactions will adhere to our notion of serializability.
 - However, if any transaction executes at a lower level, then serializability may be violated.

Potential problem with lower isolation levels

- **Dirty Read:**

- Reading a value that was written by a transaction which failed.

- **Nonrepeatable Read:**

- Allowing another transaction to write a new value between multiple reads of one transaction.
- A transaction T1 may read a given value from a table. If another transaction T2 later updates that value and T1 reads that value again, T1 will see a different value.
 - Consider that T1 reads the employee salary for Smith. Next, T2 updates the salary for Smith. If T1 reads Smith's salary again, then it will see a different value for Smith's salary.

Potential problem with lower isolation levels (2)

■ **Phantoms:**

- New rows being read using the same read with a condition.
 - 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 of T1, into the table used by T1.
 - If T1 is repeated, then T1 will see a row that previously did not exist, called a phantom.

Violation of Serializability

Type of Violation

Isolation level	Type of Violation		
	Dirty read	nonrepeatable read	phantom
READ UNCOMMITTED	yes	yes	yes
READ COMMITTED	no	yes	yes
REPEATABLE READ	no	no	yes
SERIALIZABLE	no	no	no

Database Concurrency Control

- **Purpose of Concurrency Control**
 - To enforce Isolation (through mutual exclusion) among conflicting transactions.
 - To preserve database consistency through consistency preserving execution of transactions.
 - To resolve read-write and write-write conflicts.
- **Example:**
 - In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Two-Phase Locking Techniques

- Locking is an operation which secures
 - (a) permission to Read
 - (b) permission to Write a data item for a transaction.
- Example:
 - Lock (X). Data item X is locked in behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
 - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Two-Phase Locking Techniques:

Essential components

- **Two locks modes:**
 - (a) shared (read) (b) exclusive (write).
- **Shared mode: shared lock (X)**
 - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
- **Exclusive mode: Write lock (X)**
 - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.

Two-Phase Locking Techniques:

Essential components

- **Lock Manager:**
 - Managing locks on data items.
- **Lock table:**
 - Lock manager uses it to store the identify of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Two-Phase Locking Techniques:

Essential components

- **Database requires that all transactions should be well-formed. A transaction is well-formed if:**
 - It must lock the data item before it reads or writes to it.
 - It must not lock an already locked data items and it must not try to unlock a free data item.

Lock Operation

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

Unlock Operation

$LOCK(X) \leftarrow 0$ (**unlock the item**)

if any transactions are waiting then

wake up one of the waiting the transactions;

Two-phase policy locking algorithms

Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
 - (a) **Basic**
 - (b) **Conservative**
- **Conservative:**
 - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic:**
 - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict:**
 - A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back). This is the most commonly used two-phase locking algorithm.

Dealing with Deadlock and Starvation

■ **Deadlock prevention**

- A transaction locks all data items it refers to before it begins execution.
- This way of locking prevents deadlock since a transaction never waits for a data item.
- The conservative two-phase locking uses this approach.

Dealing with Deadlock and Starvation (2)

■ Deadlock detection and resolution

- In this approach, deadlocks are allowed to happen. The scheduler maintains a wait-for-graph for detecting cycle. If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
- A wait-for-graph is created using the lock table. As soon as a transaction is blocked, it is added to the graph. When a chain like: T_i waits for T_j waits for T_k waits for T_i or T_j occurs, then this creates a cycle.

Dealing with Deadlock and Starvation (3)

■ Deadlock avoidance

- There are many variations of two-phase locking algorithm.
- Some avoid deadlock by not letting the cycle to complete.
- That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
- Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Dealing with Deadlock and Starvation (4)

■ Starvation

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Reading Material

- Fundamentals of Database Systems. Ramez Elmasri and Shamkant B. Navathe. Pearson. **Chapter 20. and Chapter 21.**

Q & A

Three light-colored wooden blocks are arranged horizontally on a dark wooden surface. The first block on the left has a black letter 'Q' on its front face. The middle block has a black ampersand '&' on its front face. The third block on the right has a black letter 'A' on its front face. The background is a soft-focus green and yellow, suggesting an outdoor setting with foliage.