Transaction Management

The Basic Concepts

ACS 575: Database Systems

Instructor: Dr. Jin Soung Yoo, Professor
Department of Computer Science
Purdue University Fort Wayne

References

- W. Lemanhieu, et al., Principles of Database Management: The Practical Guide to Storing, Managing and Analyzing Big and Small Data, Ch 14
- Jeffrey et al., Modern Database Management, Ch 7 (13th ed.) /Ch 8 (12nd ed.)

Outline

- Introduction
- ACID Properties
- Transactions and Transaction Management
- Recovery
- Concurrency Control

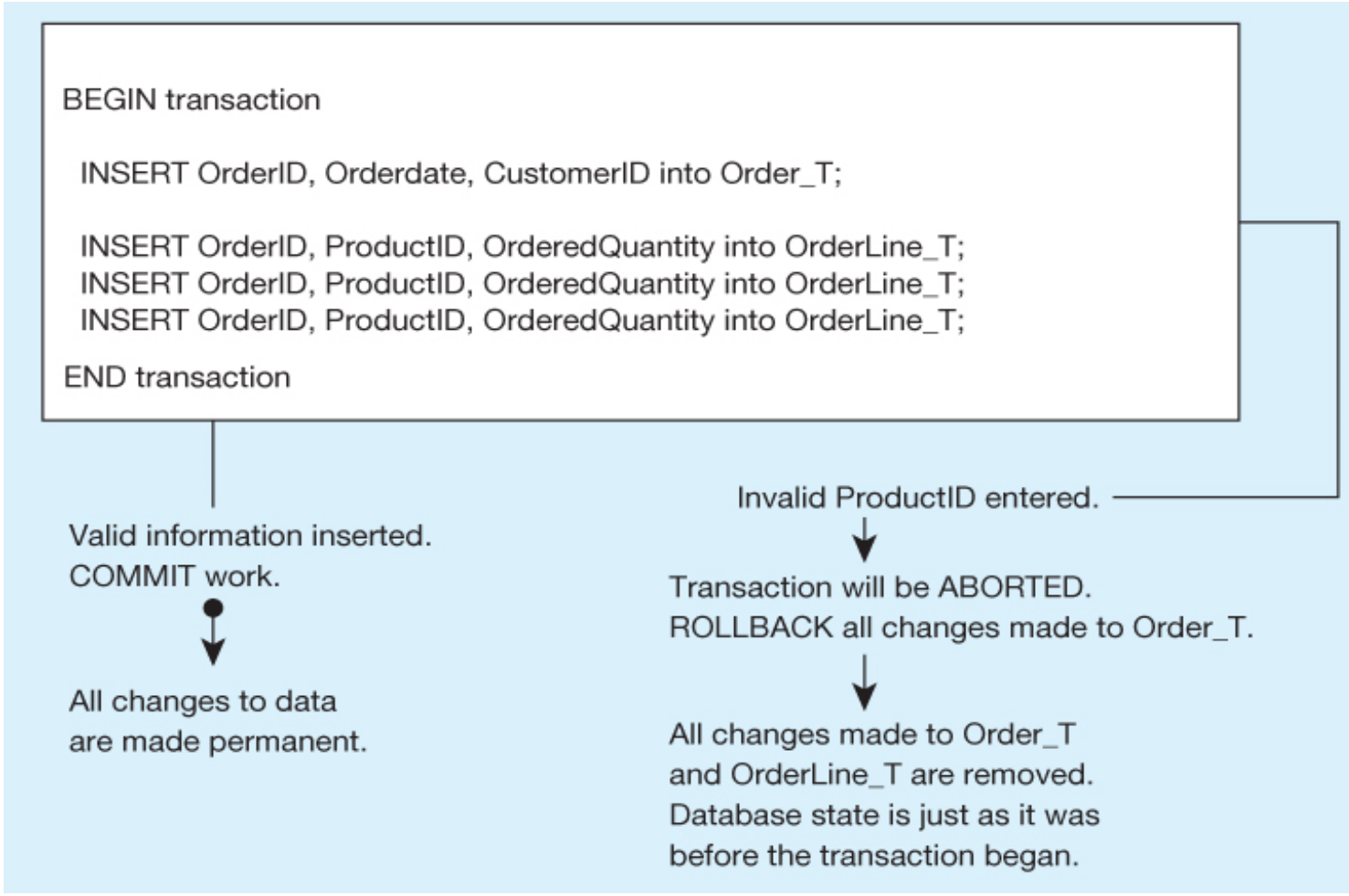
Introduction

- Majority of databases are **multi user databases**
  - Many applications and users can access the data in database in parallel.
- Concurrent access to the same data may induce different types of anomalies or unexpected problems. Errors may occur in the components of DBMS.
- As its solution to deal with such problem, DBMS supports **ACID** (Atomicity, Consistency, Isolation, Durability) properties which are the concepts of transactions and transaction management, recovery and concurrency control.

Transaction

- A **transaction** is a set of database operations (e.g., a sequence of SQL statements in RDBMS) induced by a single user or application, that should be considered as one undividable unit of work
  - E.g., A transfer between two bank accounts of the same customer includes two updates - one update to debit the first account and the other update to credit the other account for the same amount
- Transaction causes to be a database from one consistent state into another consistent state
- Transaction always ‘succeeds’ or ‘fails’ in its entirety – should never be confronted with the possible inconsistent states

Example: A SQL Sequence for Order Transaction



Recovery

- During transaction execution, different types of errors or problems may occur, e.g., hard disk failure, application/DBMS crash, division by 0, …
- For transactions that were completely successfully (i.e., committed transactions), the database system should guarantee that all changes are persisted into the database
- **Recovery** is an activity of ensuring that, whichever of the problems occurred, the database is returned to a consistent state without any data loss afterwards

Concurrency Control

- Many applications and users can access the data in database concurrently.
- They might cause inconsistencies in the data because of mutual interference
- **Concurrency control** means the coordination of transactions that execute simultaneously on the same data so that they do not cause inconsistencies in the data.

Outline

- Introduction
- ☞ **ACID Properties**
- Transactions and Transaction Management
- Recovery
- Concurrency Control

ACID Properties of Transactions

- **ACID** represents four properties (*Atomicity*, *Consistency*, *Isolation* and *Durability*) in the context of transaction management that are desirable to most conventional DBMSs.
- ACID is a set of properties of database transactions intended to guarantee validity even in the event of errors, power failures, etc.

ACID Properties

- ***Atomicity*** guarantees that multiple database operations that alter the database state can be treated as one indivisible unit of work. The transaction cannot be subdivided
- The transaction must be processed in its entirety or not at all.
  - Example: Suppose a transaction that the program accepts a new customer order, increases balance due, and stores the updated CUSTOMER record. If the new ORDER record is not inserted successfully, none of the parts of the transaction to affect the database
- This is the responsibility of the ***recovery manager***, which will induce rollbacks where necessary, by means of UNDO operations (so that no partial traces of failed transactions remain in the database)

ACID Properties

- ***Consistency*** ensures that a transaction can only bring the database from one valid state to another.
- Any data written to the database must be valid according to all defined rules. Any database constraints that must be true before the transaction must also be true after the transaction.
  - Example: If the inventory on-hand balance must be the difference between total receipts minus total issues, this will be true both before and after an order transaction, which depletes the on-hand balance to satisfy the order
- Developer (who is to ensure that the application logic that drives the transaction is flawless) is primary responsible. Also an overarching responsibility of the DBMS's transaction management system

ACID Properties

- ***Isolation*** means that, in situations where multiple transactions are executed concurrently, the outcome should be the same as if every transaction were executed in isolation (sequentially)
- Changes to the database are not revealed to users until the transaction is committed.
  - Example: This property means that other users do not know what the on-hand inventory is until an inventory transaction is complete. Other users are prohibited from simultaneously updating and possibly even reading data that are in the process of being updated.
- Responsibility of the *concurrency control mechanisms* of the DBMS, as coordinated by the ***scheduler***

ACID Properties

- ***Durability*** refers to the fact that the effects of a committed transaction should always be persisted into the database.
- Changes are permanent. Once a transaction has been committed, it will remain committed even in the case of a system failure.
- Responsibility of ***recovery manager*** (e.g. by REDO operations or data redundancy)

Outline

- Introduction
- ACID Properties
- ☞ Transactions and Transaction Management
- Recovery
- Concurrency Control

Transaction Boundary

- A database application may maintain multiple operations, and even multiple ongoing transactions, at the same time.
- DBMS is necessary to specify the transaction boundaries to delineate the transaction.
- **Transactions boundaries** can be specified implicitly or explicitly
  - Explicitly: `begin_transaction` and `end_transaction`
  - Implicitly: *from* first executable SQL statement *to* `commit`/`rollback`
  - Once the first operation is executed, the transaction is active
- If transaction completed successfully, it can be *committed*. If not, it needs to be *rolled back*.

Example: A SQL Sequence for Bank Transfer Transaction

```
<begin_transaction>

UPDATE account
SET balance = balance - :amount
WHERE accountnumber = :account_to_debit

UPDATE account
SET balance = balance + :amount
WHERE accountnumber = :account_to_credit

<end_transaction>
```

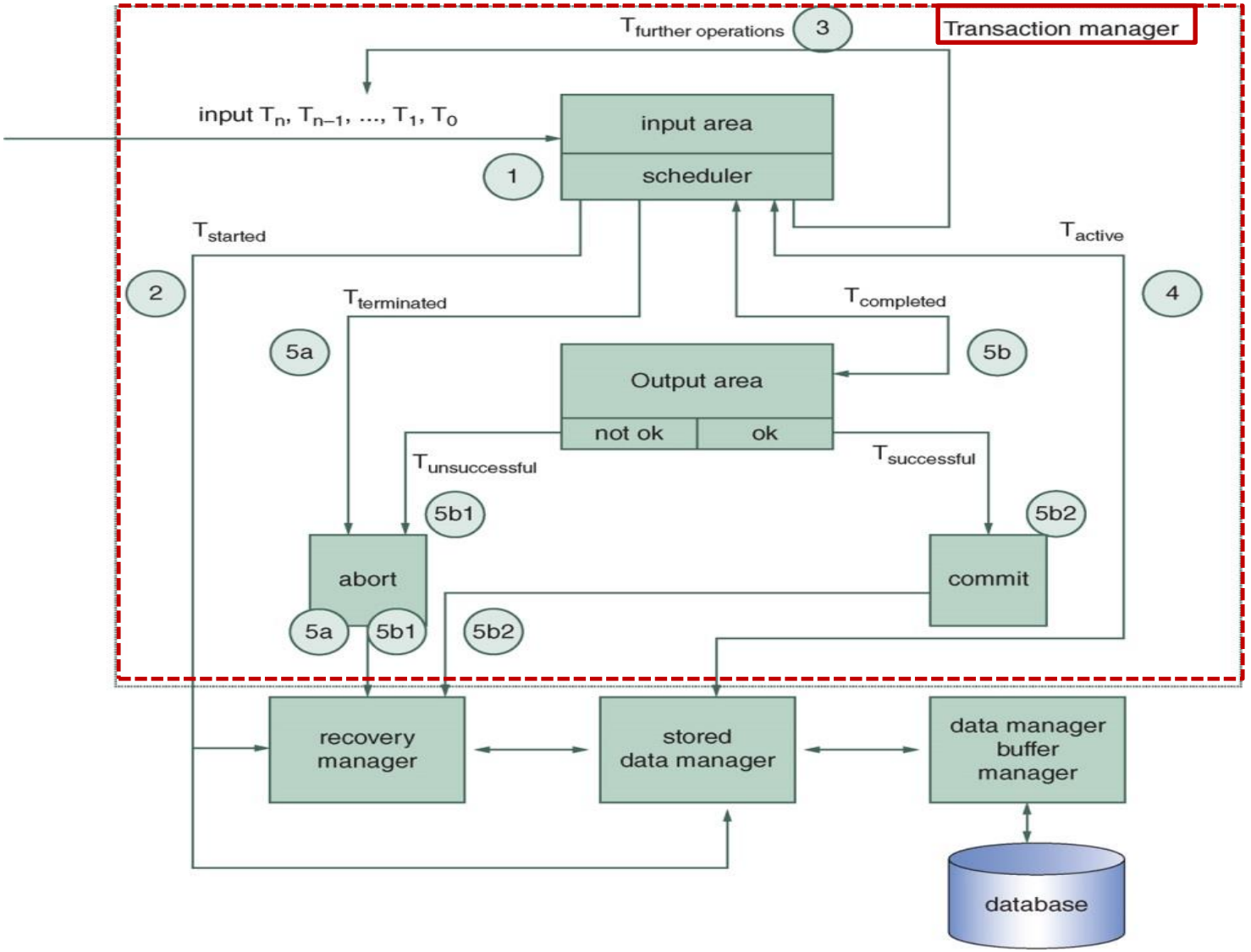
Business Transaction

- A **business transaction** is a sequence of steps that constitute some well-defined business activity
  - E.g., Admit patient in a hospital, Enter customer order in a manufacturing company
- Example: Enter customer order
  - Input the order data (keyed by the user)
  - Read the CUSOMTER record (or ask user to key it in if a new customer)
  - Accept or reject the order (if balance due plus order amount does not exceed credit limit, accept the order; otherwise, reject it).
  - If the order is accepted, increase balance due by order amount. Store the updated (or new) CUSTOMER record, Insert the accepted ORDER record in the database

Guideline for Business Transaction

- Although conceptually a transaction is a logical unit of business work, a business unit of work is divided into several database transactions.
- The general guideline is to make a database transaction as short as possible while still maintaining the integrity of the database
  - If possible, collect all user input before executing a database transaction.
  - Also to minimize the length of a transaction, check for possible errors, as early in the transaction as possible so that portions of the database can be released as soon as possible for other users if the transaction is going to be aborted.

DBMS Components Involved in Transaction Management and a Transaction Lifecyle



Transaction Management Lifecyle

- The main DBMS component responsible for coordinating transaction execution is called the ***transaction manager***.
- The transaction manager informs the *scheduler* of new transactions that were presented to its input area (1)
- As soon as a transaction can be started, the ***recovery manager*** and the ***stored data manager*** are notified (2)
- The ***scheduler*** plans the start of transactions and the execution of their respective operations. The scheduler plans their execution order (3)

Transaction Management Lifecyle (cont.)

- Upon execution, the operations trigger interaction with the database through the *stored data manager* (4)
- The ***stored data manager*** coordinates the I/O instruction and hence the physical interactions with the database files.
- The *stored data manager* also interacts with **the *buffer manager*** which is responsible for data exchange between the database buffer in memory and the physical database files.

Transaction Management Lifecyle (cont.)

- If the transaction was completed unsuccessfully, because a problem was detected (5b1) or if the transaction was terminated before completion (5a), the transaction reaches an aborted state.
- If any (interminated) changes were already written to disk, they is undone by the ***recovery manager***
- When a transaction was completed successfully (5b2), the transaction reaches the committed state.
- Updates as induced by the transaction, can be considered as permanent and should be persisted into the physical database files. The ***recovery manager*** is also invoked to coordinate this.

Logfile in Transaction Management

- The ***logfile*** is a vital element in transaction management and recovery. The logfile registers:
  1. a unique log sequence number
  2. a unique transaction identifier
  3. a marking to denote the start of a transaction, along with the transaction's start time and indication whether the transaction is read only or read/write
  4. identifiers of the database records involved in the transaction, as well as the operation(s) they were subjected to
  5. ***before images*** (original values) of all records that participated in the transaction
  6. ***after images*** (new values) of all records that were changed by the transaction
  7. the current state of the transaction (*active*, *committed* or *aborted*)

Write Ahead Log

- Transaction management uses ***write ahead log strategy***.

  - All updates are registered on the logfile before written to disk.

- In this way, “before images” are always recorded on the logfile prior to the actual values being overwritten in the physical database files

Checkpoints

- Checkpoints are relevant to the write ahead log strategy.
- Logfile may also contain ***checkpoints*** (synchronization points) - moments when buffered updates by active transactions, as present in the database buffer, are written to disk at once

Outline

- Introduction
- ACID Properties
- Transactions and Transaction Management
- **Recovery**
  - System Recovery
  - Media Recovery
- Concurrency Control

Types of Failures

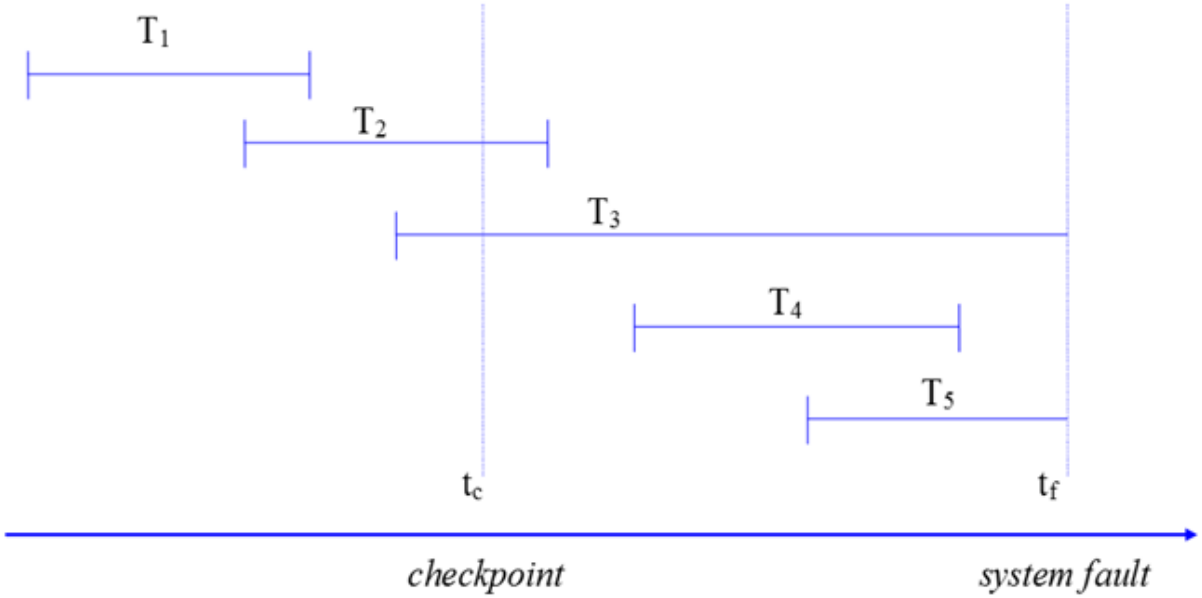
- **Transaction failure** results from an error in the logic that drives the transaction's operations (e.g., wrong input, uninitialized variables) and/or in the application logic
- **System failure** occurs if the operating system or the database system crashes
- **Media failure** occurs if the secondary storage is damaged or inaccessible

System Recovery

- In case of system failure, there are **two types of transactions:**
  - transactions already reached the committed state before failure
  - transactions still in an active state
- **Logfile** is essential to take account of which updates were made by which transactions (and when) and to keep track of before images and after images needed for the UNDO and REDO
  - **UNDO** is for updates already written to disk
  - **REDO** is for changes still pending in the database buffer upon system failure.

Example of Transactions and System Recovery

- Figure presents five transactions ($\mathbf{T_1}$ to $\mathbf{T_5}$) that are executed more or less simultaneously. Let's assume for now that the transactions do not interfere; we will cover concurrency control later.
- Suppose a checkpoint was registered on the logfile at time $\mathbf{T_C}$, marking the last time when pending updates in the database buffer were persisted into the physical database files.
- Later, at time $\mathbf{T_f}$, a system fault occurred, resulting in loss of the database buffer.



Result of Transaction and System Recovery

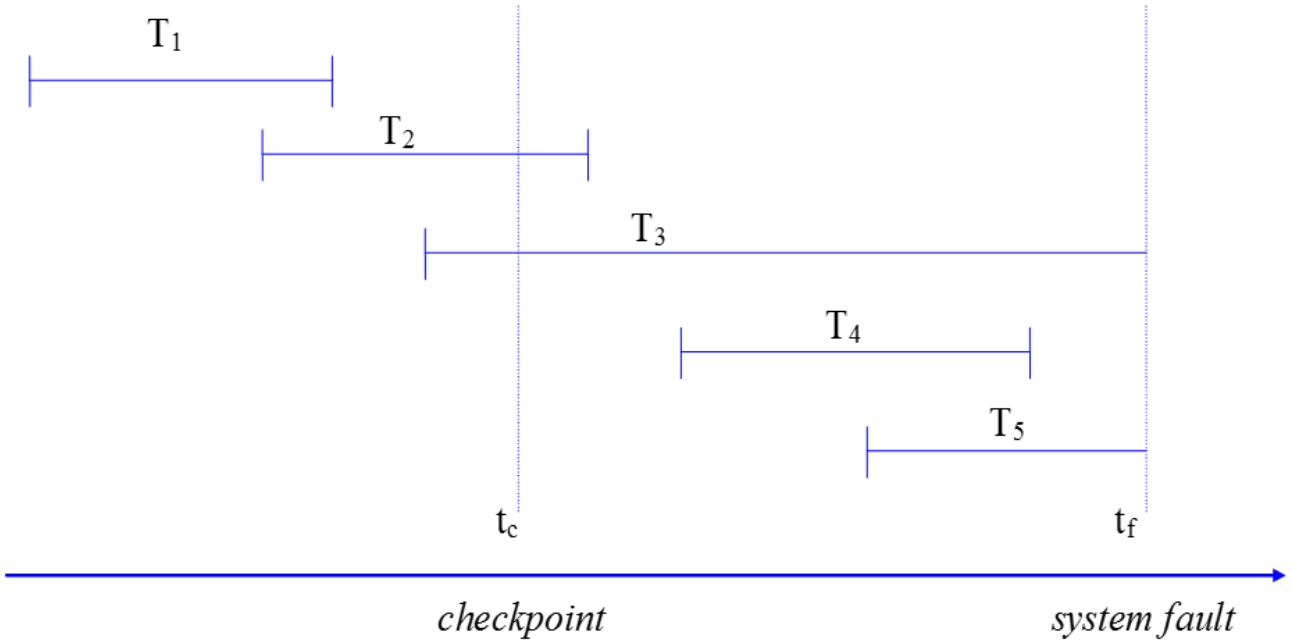
T_1: nothing

T_2: REDO

T_3: UNDO

T_4: REDO

T_5: nothing

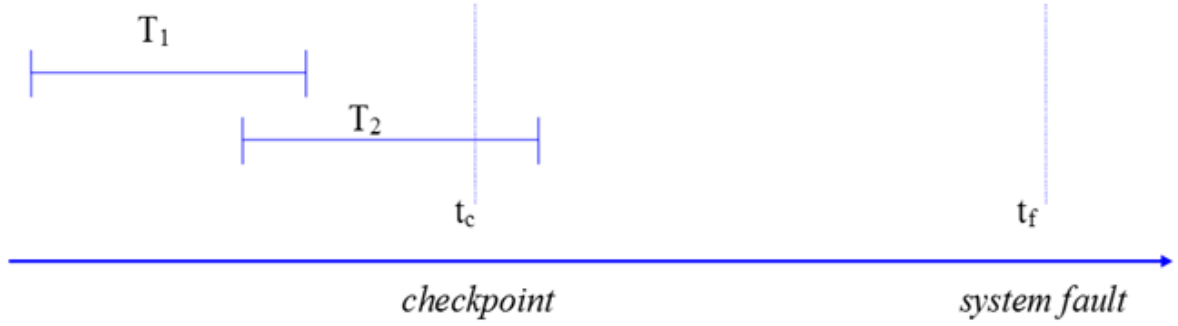


Transactions and System Recovery

- **Transaction T_1** committed before time T_C , when the checkpoint was registered. For this transaction, no recovery operations are required. Indeed, all the transaction's updates were already written to disk, so a REDO is not necessary. Moreover, since the transaction was successfully committed, no UNDO operations are required either.
- **Transaction T_2** was still active on time T_C. However, the transaction committed successfully before the actual system fault at time T_f. Since the transaction committed, but not all updates were written to disk before the database buffer was lost, a REDO is required to eventually persist all the transaction's updates.

T_1: nothing

T_2: REDO

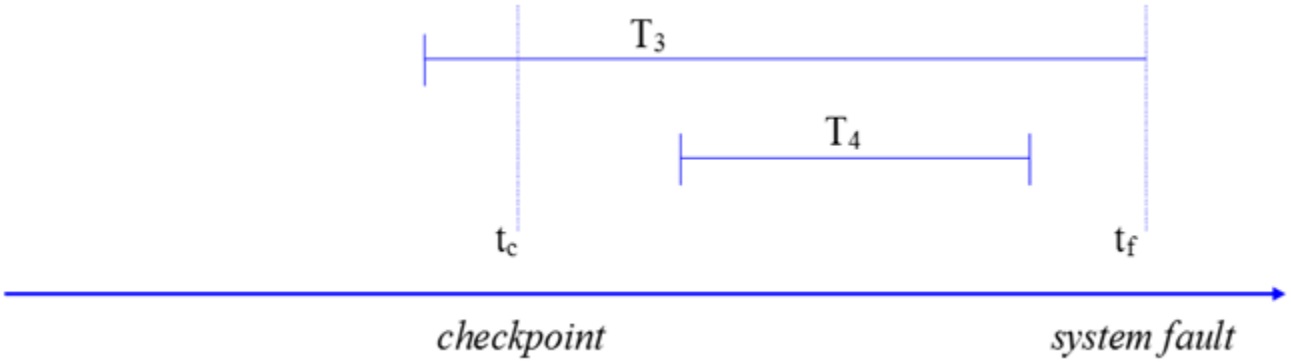


Transactions and System Recovery

- **Transaction T_3** was still active when the system fault occurred at time T_f. Some of its updates were still pending in the database buffer and disappeared by themselves when the buffer was lost. However, other updates, the ones made before T_C , were already written to disk. Since the transaction cannot continue because of the system fault, it cannot commit and must be aborted. The partial changes already written to disk must be rolled back by means of UNDO operations.
- **Transaction T_4** started after time T_C and committed before the system fault at time T_f. Since all of its updates were still pending in the database buffer, a REDO is required to persist its effects into the database.

T_3: UNDO

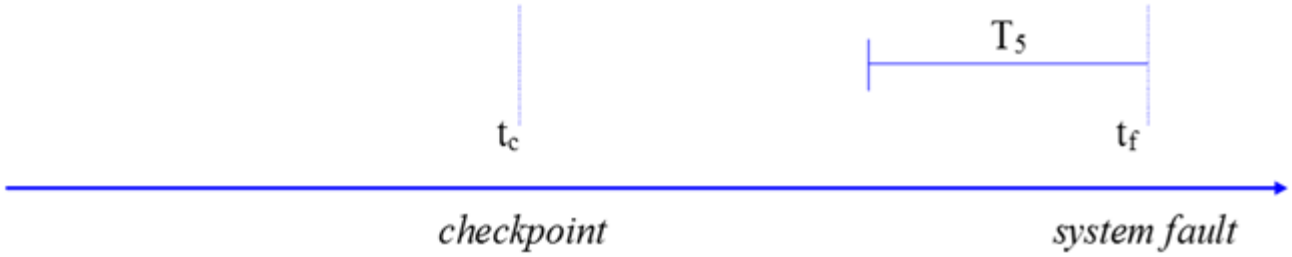
T_4: REDO



Transactions and System Recovery

- **Transaction T_5** also started after T_C. Since the transaction was still active at time T_f, it cannot continue because of the system fault and will be aborted. However, since all its updates were still pending in the database buffer, they disappear along with the rest of the buffer's contents. No UNDO is required.

T_5: nothing



Media Recovery

- **Media recovery** is required if the physical database files and/or the logfile are damaged due to a malfunction of the storage media or the storage system.
- Media recovery is invariably based on some type of data redundancy stored on offline (e.g., a tape vault) or online media (e.g., online backup hard disk drive)
- Tradeoff between cost to maintain the redundant data and time needed to restore the system
- Two types: disk mirroring and archiving

Outline

- Introduction
- ACID Properties
- Transactions and Transaction Management
- Recovery
- **Concurrency Control**
  - Typical Concurrency Problems
  - Schedules and Serial Schedules
  - Serializable Schedules
  - Locking and Locking Protocols

Controlling Concurrent Access

- Databases are shared resources.
- When more than one transaction is being processed against a database at the same time, the transactions are considered to be *concurrent*.
- The actions that must be taken to ensure that data integrity is maintained are called ***currency control actions***.
  - If users are only reading data, no data integrity problems will be encountered.
- The currency control actions are implemented by a DBMS. But a DBA must understand these actions.

Typical Concurrency Problems

- ***Scheduler*** is responsible for planning the execution of transactions and their operations
- Simple serial execution for multiple transactions would be very inefficient
- *Scheduler* will ensure that operations of the transactions can be executed in an interleaved way, with the CPU switching among the transactions
- But interference problems could occur problems such as
  - **lost update problem**
  - **uncommitted dependency problem**
  - **inconsistent analysis problem**

Lost Update Problem

- The most common problem encountered when multiple users attempt to update a database without adequate concurrency control is ‘lost updates’
- **Lost update** problem occurs if an otherwise successful update of a data item by a transaction is overwritten by another transaction that wasn’t ‘aware’ of the first update

Example - Lost Updates

- John and Marsha have a joint checking account, and both want to withdraw some cash at the same time, each using an ATM terminal in a different location.
- John's transaction (T2) reads the account balance (which is $100), and he proceeds to deposit $120.
- Before the transaction writes the new account balance ($220), Marsha's transaction (T1) reads the account balance (which is still $100).
- Marsha then withdraw $50, leaving a balance of $50.

time	T_1	T_2	$amount_x$
t_1		begin transaction	100
t_2	begin transaction	read($amount_x$)	100
t_3	read($amount_x$)	$amount_x = amount_x + 120$	100
t_4	$amount_x = amount_x - 50$		

Example - Lost Updates (cont.)

time	T_1	T_2	$amount_x$
t_1		begin transaction	100
t_2	begin transaction	read($amount_x$)	100
t_3	read($amount_x$)	$amount_x$ = $amount_x$ + 120	100
t_4	$amount_x$ = $amount_x$ - 50	write($amount_x$)	220
t_5	write($amount_x$)	commit	50
t_6	commit		50

The update by T_2 is completely lost !!

- John transaction (T2) then write this account, which replaces the one written by Marsha's transaction (T1).
- Marsha transaction (T1) then write this account, which replaces the one written by John's transaction (T2).
- The effect of John's update (T2) has been lost due to interference between the transactions, and the bank is unhanppy.

Uncommitted Dependency Problems

- If a transaction reads one or more data items that are being updated by another, as yet uncommitted, transaction, we may run into the **uncommitted dependency** (a.k.a. **dirty read**) problem

time	T_1	T_2	$amount_x$
t_1		begin transaction	100
t_2		read($amount_x$)	100
t_3		$amount_x = amount_x + 120$	100
t_4	begin transaction	write($amount_x$)	220
t_5	read($amount_x$)		220
t_6	$amount_x = amount_x - 50$	rollback	100
t_7	write($amount_x$)		170
t_8	commit		170

T_1 has read and possibly used "non existent" value of amount !!

Inconsistency Analysis Problems

- The **inconsistent analysis** problem denotes a situation where a transaction reads partial results of another transaction that simultaneously interacts with (and updates) the same data items.

time	T_1	T_2	$amount_x$	$_y$	$_z$	*sum*
t_1		begin transaction	100	75	60	
t_2	begin transaction	sum = 0	100	75	60	0
t_3	read($amount_x$)	read($amount_x$)	100	75	60	0
t_4	$amount_x = amount_x - 50$	sum = sum + $amount_x$	100	75	60	100
t_5	write($amount_x$)	read($amount_y$)	50	75	60	100
t_6	read($amount_z$)	sum = sum + $amount_y$	50	75	60	175
t_7	$amount_z = amount_z + 50$		50	75	60	175
t_8	write($amount_z$)		50	75	110	175
t_9	commit	read($amount_z$)	50	75	110	175
t_{10}		sum = sum + $amount_z$	50	75	110	285
t_{11}		commit	50	75	110	285

The \$50 transferred by T_1 is included twice in the sum calculated by T_2; the result is 285 instead of 235!!

Other Concurrency Problems

- **Nonrepeatable read** (**unrepeatable read**) occurs when a transaction T_1 reads the same row multiple times, but obtains different subsequent values, because another transaction T_2 updated this row in the meantime
- **Phantom reads** can occur when a transaction T_2 is executing insert or delete operations on a set of rows that are being read by a transaction T_1

Outline

- Introduction
- ACID Properties
- Transactions and Transaction Management
- Recovery
- Concurrency Control
  - Typical Concurrency Problems
  - ☞ **Schedules and Serial Schedules**
  - Serializable Schedules
  - Locking and Locking Protocols

Schedule

- A ***schedule*** S is defined as a set of *n* transactions, and a sequential ordering over the statements of these transactions, for which the following property holds:

  "For each transaction T that participates in a schedule S and for all statements s_i and s_j that belong to the same transaction T: if statement s_i precedes statement s_j in T, then s_i is scheduled to be executed before s_j in S."

- Schedule preserves the ordering of the individual statements *within* each transaction but allows an arbitrary ordering of statements *between* transactions

Serial Schedule

- Schedule S is ***serial*** if all statements s_i of the same transaction T are scheduled consecutively, without any interleave with statements from a different transaction
- But serial schedules prevent parallel transaction execution
- We need a non-serial, correct schedule!!

Serializable Schedules

- A ***serializable*** schedule is a non-serial schedule which is *equivalent* to a serial schedule
- Two schedules S_1 and S_2 (with the same transactions T_1, T_2, ..., T_n) are **equivalent** if
  1. *For each operation $read_x$ of T_i in S_1, the following holds: if a value x that is read by this operation, was last written by an operation $write_x$ of a transaction T_j in S_1, then that same operation $read_x$ of T_i in S_2 should read the value of x, as written by the same operation $write_x$ of T_j in S_2*
  2. *For each value x that is affected by a write operation in these schedules, the following holds: the last write operation $write_x$ in schedule S_1, as executed as part of transaction T_i, should also be the last write operation on x in schedule S_2, again as part of transaction T_i.*

Examples

	schedule S_1 serial schedule		*schedule S_2 non serial schedule*	
time	T_1	T_2	T_1	T_2
t_1	begin transaction		begin transaction	
t_2	read($amount_x$)		read($amount_x$)	
t_3	$amount_x$ = $amount_x$ + 50		$amount_x$ = $amount_x$ + 50	
t_4	write($amount_x$)		write($amount_x$)	begin transaction
t_5	read($amount_y$)			read($amount_x$)
t_6	$amount_y$ = $amount_y$ - 50			$amount_x$ = $amount_x$ x 2
t_7	write($amount_y$)			write($amount_x$)
t_8	end transaction			read($amount_y$)
t_9				$amount_y$ = $amount_y$ x 2
t_{10}		begin transaction		write($amount_y$)
t_{11}		read($amount_x$)	read($amount_y$)	end transaction
t_{12}		$amount_x$ = $amount_x$ x 2	$amount_y$ = $amount_y$ - 50	
t_{13}		write($amount_x$)	write($amount_y$)	
t_{14}		read($amount_y$)	end transaction	
t_{15}		$amount_y$ = $amount_y$ x 2		
t_{16}		write($amount_y$)		
t_{17}		end transaction		

The read (amount) operation of T_2 in S_1 reads the value of $amount_y$ as written by T_1, whereas the same operation in S_2 reads the original value of $amount_y$

S_1 is not equivalent to S_2 because of violation of condition 1. No exist any other serial schedule that is equivalent to S_2

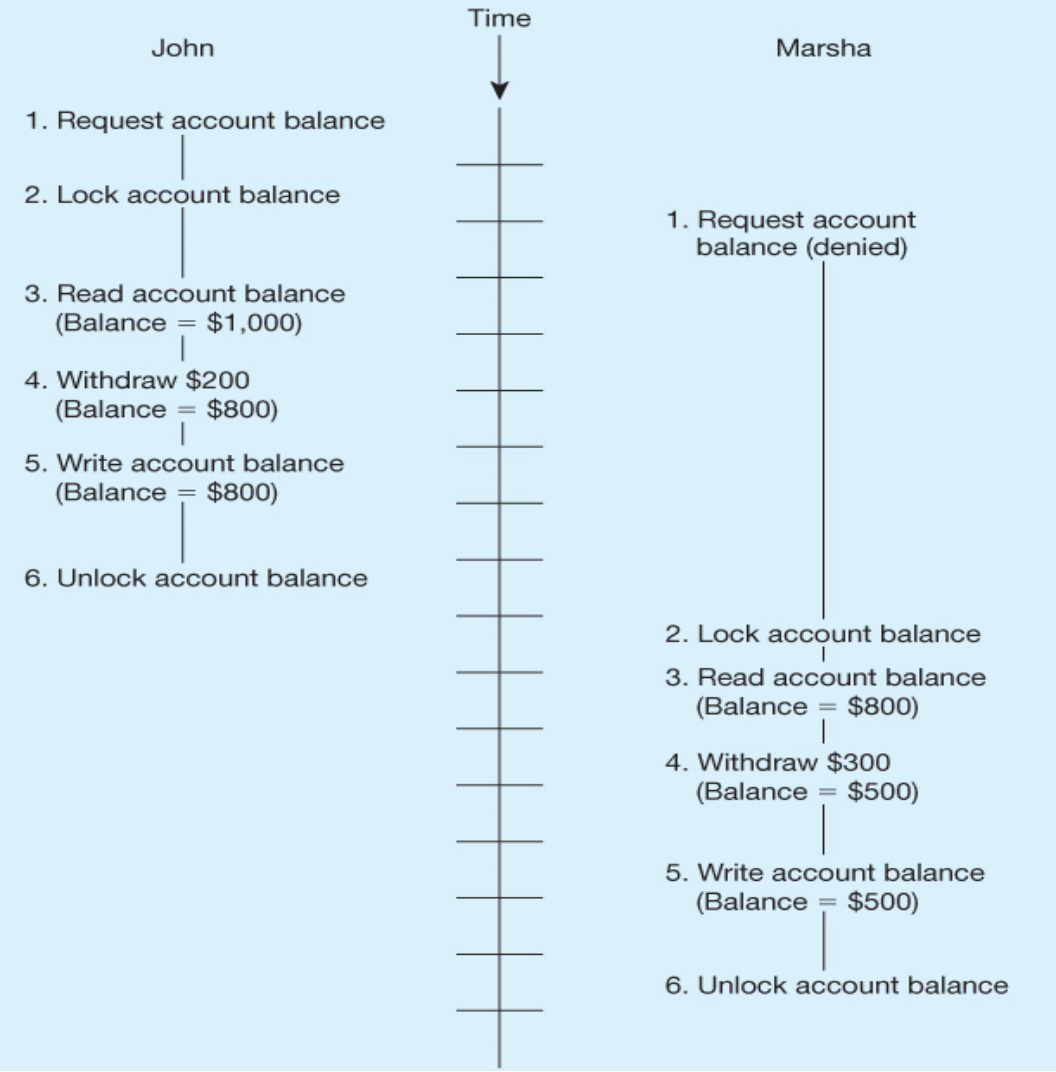
Outline

- Introduction
- ACID Properties
- Transactions and Transaction Management
- Recovery
- Concurrency Control
  - Typical Concurrency Problems
  - Schedules and Serial Schedules
  - Serializable Schedules
  - ☞ **Locking and Locking Protocols**

Locking Mechanisms

- **Purpose of *locking*** is to ensure that, in situations where different concurrent transactions attempt to access the same database object, access is only granted in such a way that no conflicts can occur
- With **locking**, any data that are retrieved by a user for updating must be locked, or denied for other users, until the update is complete or aborted
  - Locking data is much like checking a book out of the library, it is unavailable to others until the borrower returns it.
- The locking mechanism enforces a sequential updating process that prevents erroneous update.

Example: Updates with Locking



Suppose John and Marsha have a joint checking account.

Marsha's transaction cannot access the account record until **John**'s transaction has returned the updated record to the database and unlocked the record.

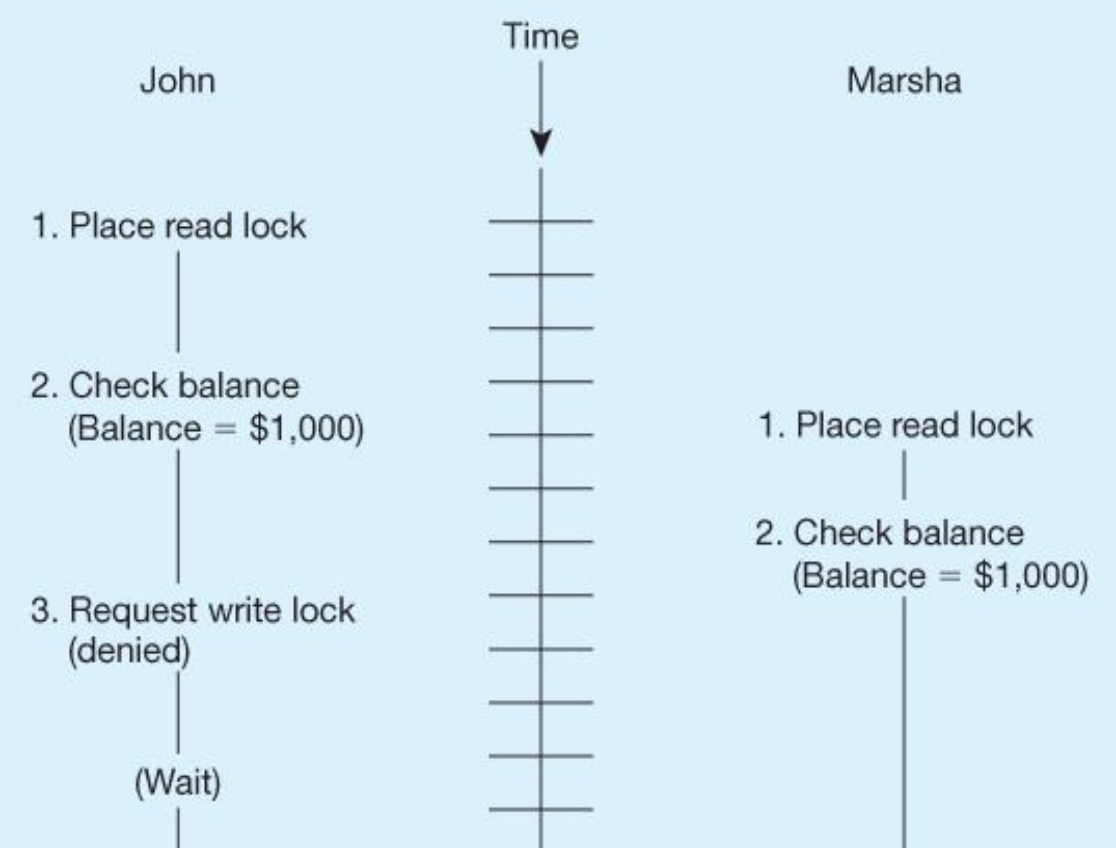
Lock Manager

- **Lock manager** is responsible for granting locks (*locking*) and releasing locks (*unlocking*) by applying a *locking protocol*
- **Locking protocol** is set of rules to determine what locks can be granted in what situation (based on e.g. compatibility matrix)
- Lock manager also uses a **lock table**
  - which locks are currently held by which transaction, which transactions are waiting to acquire certain locks, etc.
- Lock manager needs to ensure ‘fairness’ of transaction scheduling, e.g., to avoid starvation

Locking Type

- An **exclusive lock** (*x*-lock or write lock) means that a single transaction acquires the sole privilege to interact with that specific database object at that time
- With exclusive locking, no other transactions are allowed to read or write the same object.
- A **shared lock** (*s*-lock or read lock) guarantees that no other transactions will update that same object for as long as the lock is held
- With shared locking, other transactions may hold a shared lock on that same object as well, however they are only allowed to read it

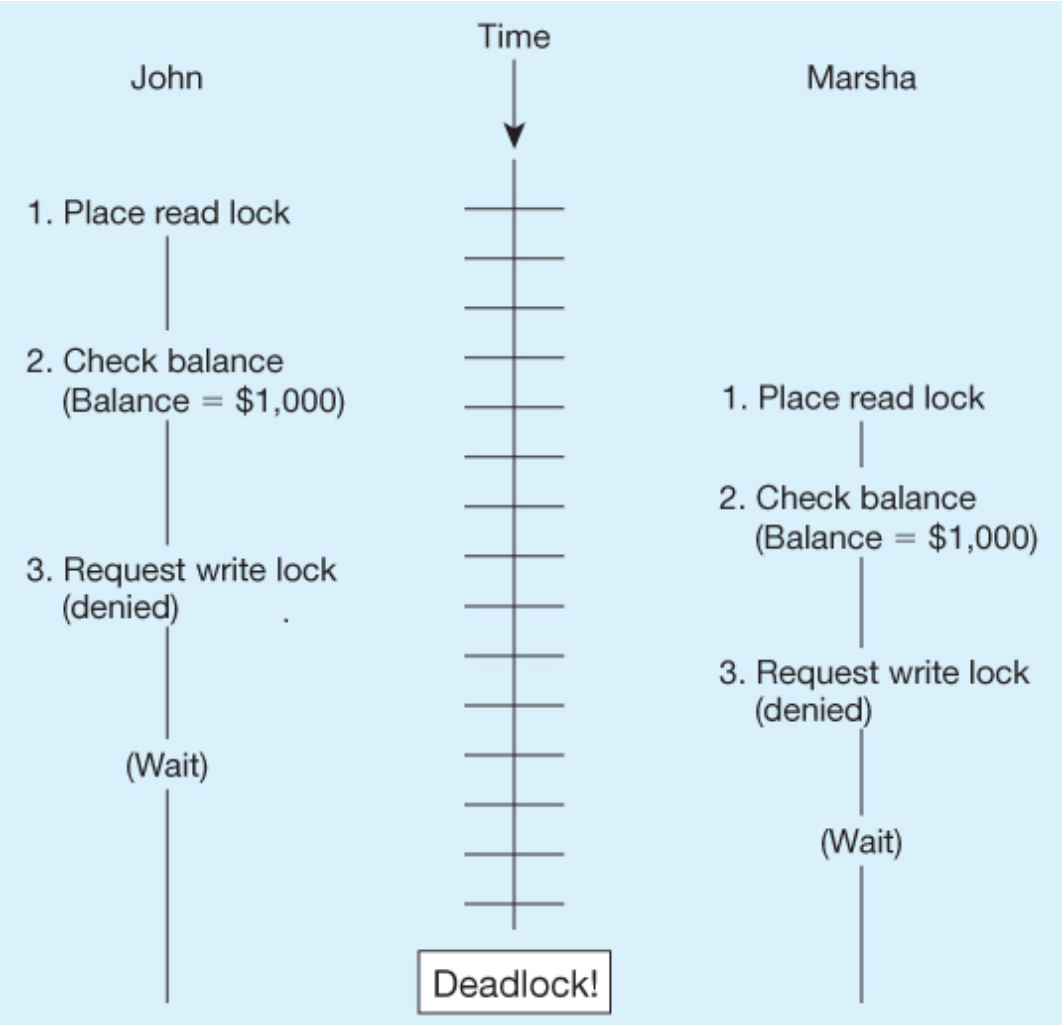
Example: Use of Shared and Exclusive Locks



- When **John** initiates his transaction, the program places a read lock on his account record
- When **John** requests a withdrawal, the program attempts to place an exclusive lock.
- However, **Marsh** has already initiated a transaction that has placed a read lock on the same record. As result, **John**'s request is denied.

Deadlock

- When two or more transactions have locked a common resource and each must wait for the other to unlock the resource, **deadlock** happens.



Managing Deadlock

- Two basic ways to resolve deadlocks
  - **Deadlock resolution –** This allows deadlocks to occur but to build mechanisms into a DBMS for detecting and breaking the deadlocks
  - **Deadlock prevention** – When deadlock prevention is employed, user programs must lock all records they will require at the beginning of a transaction rather than one at a time.

Locking Protocol

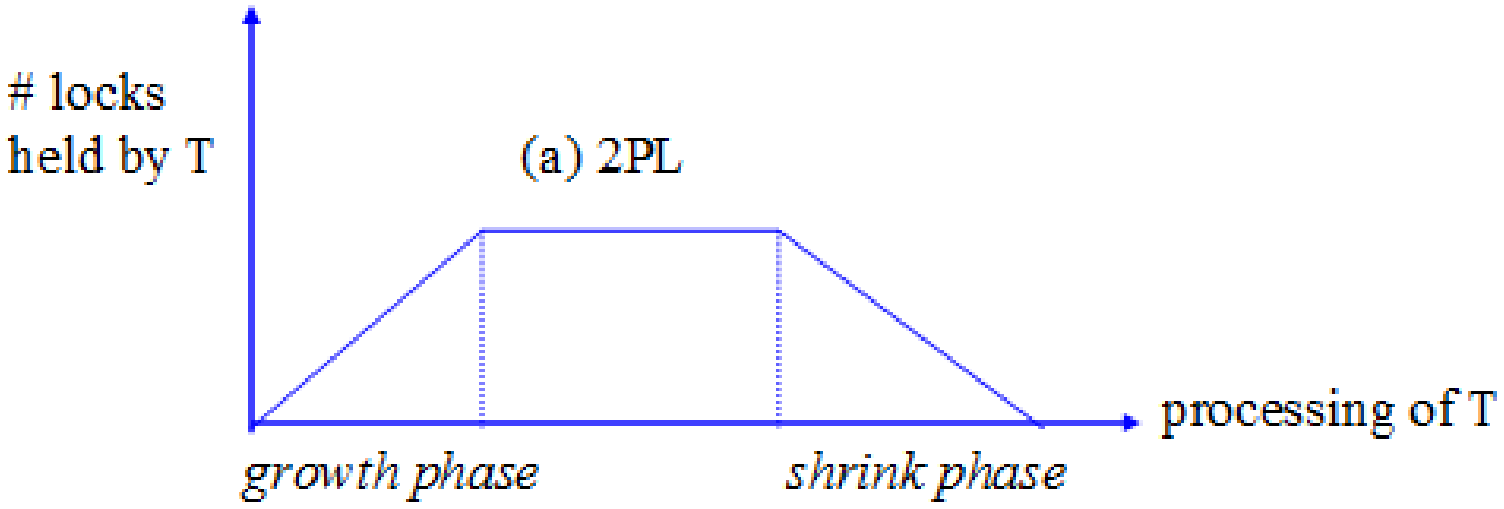
- **Locking protocol** is set of rules to determine what locks can be granted in what situation (based on e.g. compatibility matrix)
- There are many locking protocols, but the most well known in a standalone database context is **Two-Phase Locking**
- Where all locking operations necessary for a transaction occur before any resources are unlocked, a *two-phase locking protocol* is being used.

Two-Phase Locking (2PL) Protocol

- **Two-Phase Locking (2PL)** protocol works as follows:
  1. Before a transaction can read (/update) a database object, it should acquire **a shared (/exclusive) lock** on that object
  2. Lock manager determines if requested locks can be granted, based on compatibility matrix
  3. Acquiring and releasing locks occurs in 2 phases
     - **growth phase**: locks can be acquired but no locks can be released
     - **shrink phase**: locks are gradually released, and no additional locks can be acquired

Two-Phase Locking (2PL) Protocol

- According to the basic 2PL protocol, a transaction can already start releasing locks before it has attained the "committed" state, on the condition that no further locks are acquired after releasing the first lock.



2PL and Concurrency Control

- When applying the 2PL Protocol, concurrency problems are avoided. - 2PL is a concurrency control method that guarantees serializability
- E.g., How to solve the lost update problem with locking

time	T_1	T_2	$amount_x$
t_1		begin transaction	100
t_2	begin transaction	x-lock($amount_x$)	100
t_3	x-lock($amount_x$)	read($amount_x$)	100
t_4	wait	$amount_x = amount_x + 120$	100
t_5	wait	write($amount_x$)	220
t_6	wait	commit	220
t_7	wait	unlock($amount_x$)	220
t_8	read($amount_x$)		220
t_9	$amount_x = amount_x - 50$		220
t_{10}	write($amount_x$)		170
t_{11}	commit		170
t_{12}	unlock($amount_x$)		170

- T_1 has to wait until $amount_x$ is unlocked by T_2 before T_1 can acquire a write lock on it.
- As a consequence, T_1 is now aware of the update of $amount_x$ by T_2 and, consequently, the lost update problem is resolved

Lock Granularity (Locking Level)

- The locking level (lock granularity) is the extent of the database resource that is included with each lock.
- Common lock levels are:
  - **Database**: The entire database is locked and becomes unavailable to other users. This level has limited application, such as during a backup of the entire database
  - **Table**: The entire table containing a requested record is locked. This level is appropriate mainly for bulk updates that will update the entire table, such as giving all employees a two percent raise
  - **Block or page**: The physical storage block (or page) containing a requested record is locked. This level is the most commonly implemented locking level.

Lock Granularity (cont.)

- Common lock levels (cont.)
  - **Record**: Only the requested record (or **row**) is locked. All other records, even within a table, are available to other users. Row-level locking uses more memory but allows greater concurrency, which works better in multi-user systems.
  - **Field**: Only the particular field (or **column**) in a requested record is locked. This level may be appropriate when most updates affect only one or two fields in a record. Field-level locks require considerable overhead and are seldom used.

Lock Granularity

- Trade-off between locking overhead and transaction throughput
- Locking at a fine-grained level (e.g., an individual tuple) has the least negative impact on throughput.
- Many DBMSs provide the option to have the optimal granularity level determined by the database system

Conclusions

- Terms: Transactions, Recovery and Concurrency Control
- ACID Properties of Transactions
- Transactions and Transaction Management
- Recovery
  - System Recovery
  - Media Recovery
- Concurrency Control
  - Typical Concurrency Problems
  - Schedules and Serial Schedules
  - Serializable Schedules
  - Locking and Locking Protocols

Appendix: Oracle Transaction Management

- https://docs.oracle.com/en/database/oracle/oracle-database/19/cncpt/data-concurrency-and-consistency.html#GUID-E8CBA9C5-58E3-460F-A82A-850E0152E95C