

CS315: DATABASE SYSTEMS TRANSACTIONS AND RECOVERY SYSTEMS

Arnab Bhattacharya

`arnabb@cse.iitk.ac.in`

Computer Science and Engineering,
Indian Institute of Technology, Kanpur

<http://web.cse.iitk.ac.in/~cs315/>

2nd semester, 2018-19

Mon 12:00-13:15, Tue 9:00-10:15

ACID Properties

- A **transaction** is a logical unit of a program
- To preserve data integrity, a database must follow four properties
 - 1 **Atomicity**: Either all operations of a transaction are reflected or none are reflected
 - 2 **Consistency**: If a database is consistent before the execution of the transaction, it must be consistent after it
 - 3 **Isolation**: Although multiple transactions may execute concurrently, each transaction must be unaware of others, i.e., to a transaction, it must seem that either any other transaction has completed execution or has not started execution at all
 - 4 **Durability**: After a transaction finishes successfully, the changes must be permanent in the database despite subsequent failures
- Together, these four properties are called the **ACID** properties

Transaction Model

- A **transaction** is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity

Transaction Model

- A **transaction** is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity
- Two basic operations: **read** and **write**
- **read(x)**: reads a database item named *x*
- **write(x)**: writes to a database item named *x*
- Database program uses the same name *x* for the variable

Transaction Model

- A **transaction** is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity
- Two basic operations: **read** and **write**
- **read(x)**: reads a database item named x
- **write(x)**: writes to a database item named x
- Database program uses the same name x for the variable
- A transaction can read or write a data item *only once*
- Conflicts with read and write

Transaction Model

- A **transaction** is a logical unit of a program that reads and writes various named data items
- Thus, a database is simply a collection of *named items*
- *Granularity* of data: field, record, block, relation
- Transaction concepts are independent of granularity
- Two basic operations: **read** and **write**
- **read(x)**: reads a database item named *x*
- **write(x)**: writes to a database item named *x*
- Database program uses the same name *x* for the variable
- A transaction can read or write a data item *only once*
- Conflicts with read and write
 - **RAW**: read-after-write
 - **WAR**: write-after-read
 - **WAW**: write-after-write

Example

- Transfer 50 rupees from account a to account b
- `read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)`
- Atomicity:

Example

- Transfer 50 rupees from account a to account b
- `read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)`
- Atomicity: a must not be debited without crediting b
- Consistency:

Example

- Transfer 50 rupees from account a to account b
- `read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)`
- Atomicity: a must not be debited without crediting b
- Consistency: Sum of a and b remains constant
- Isolation:

Example

- Transfer 50 rupees from account a to account b
- `read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)`
- Atomicity: a must not be debited without crediting b
- Consistency: Sum of a and b remains constant
- Isolation: If another transaction reads a and b , it must see one of the two states: before or after the transaction; otherwise $a + b$ is wrong
- Durability:

Example

- Transfer 50 rupees from account a to account b
- `read(a); a := a - 50; write(a); read(b); b := b + 50; write(b)`
- Atomicity: a must not be debited without crediting b
- Consistency: Sum of a and b remains constant
- Isolation: If another transaction reads a and b , it must see one of the two states: before or after the transaction; otherwise $a + b$ is wrong
- Durability: If b is notified of credit, it must persist even if the database crashes

Problems in Transactions

- **Lost update:** Update by one transaction is overwritten by another transaction

Problems in Transactions

- **Lost update**: Update by one transaction is overwritten by another transaction
- **Temporary update** or **Dirty read**: A transaction updates and then fails; another transaction reads it before it is reverted to original value

Problems in Transactions

- **Lost update**: Update by one transaction is overwritten by another transaction
- **Temporary update** or **Dirty read**: A transaction updates and then fails; another transaction reads it before it is reverted to original value
- **Incorrect summary**: One transaction is updating values while other is computing an aggregate on them

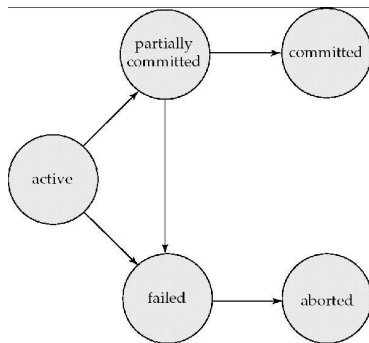
Causes of Transaction Failures

Causes of Transaction Failures

- System crash
 - Memory is lost
- System error
 - Divide by zero
- Exceptions
 - Insufficient account balance
- Concurrency enforcement
 - Deadlock detection
- Disk crash
 - Persistency fails
- Physical problems
 - Power failure, fire

States of a Transaction

- **Active**: transaction is executing
- **Partially committed**: after last statement has been executed
- **Failed**: when execution cannot proceed
- **Committed**: after successful completion
- **Aborted**: transaction has been rolled back and it has been ensured that there is no effect of the transaction



Shadow Database Scheme

- **Recovery management system** of a database ensures that atomicity and durability properties are maintained
- **Shadow database** scheme enforces atomicity
 - A *shadow copy* of the database is made before any transaction
 - All updates are made on the shadow copy
 - If the transaction finishes successfully, the database pointer is updated to the shadow copy
 - If the transaction fails, the old database pointer which points to the original database is maintained
- Inefficient for large databases
- Cannot efficiently handle concurrent transactions

- **Log** or **journal** keeps track of all transaction operations
- Log enforces atomicity and durability
 - Log is maintained on disk
 - Log is periodically backed up to archival storage to guard against disk failures
- For a transaction T , following log records are maintained
 - (start, T)
 - (write, T , x , old, new)
 - (read, T , x , val)
 - (commit, T)
 - (abort, T)

- **Log** or **journal** keeps track of all transaction operations
- Log enforces atomicity and durability
 - Log is maintained on disk
 - Log is periodically backed up to archival storage to guard against disk failures
- For a transaction T , following log records are maintained
 - (start, T)
 - (write, T , x , old, new)
 - (read, T , x , val)
 - (commit, T)
 - (abort, T)
- Recovery using logs
 - **Undo** possible by traversing log *backward* and setting database items to *old* values
 - **Redo** possible by traversing log *forward* and setting database items to *new* values

Commit Point

- A transaction reaches its **commit point** when all its operations have been executed successfully and they have been recorded in the log
- Beyond the commit point, the transaction is said to be *committed*, and its effect on the database is assumed to be permanent
- A *commit entry* (commit, T) is made in the log

Commit Point

- A transaction reaches its **commit point** when all its operations have been executed successfully and they have been recorded in the log
- Beyond the commit point, the transaction is said to be *committed*, and its effect on the database is assumed to be permanent
- A *commit entry* (commit, T) is made in the log
- A transaction needs to **rollback** if there is a (start, T) entry but no (commit, T) entry
 - It may also need to be rolled back if there is a (abort, T) entry
 - Undo operations need to be performed

Commit Point

- A transaction reaches its **commit point** when all its operations have been executed successfully and they have been recorded in the log
- Beyond the commit point, the transaction is said to be *committed*, and its effect on the database is assumed to be permanent
- A *commit entry* (commit, T) is made in the log
- A transaction needs to **rollback** if there is a (start, T) entry but no (commit, T) entry
 - It may also need to be rolled back if there is a (abort, T) entry
 - Undo operations need to be performed
- Before a transaction reaches its commit point, any portion of the log not yet written to disk must be flushed – this is called **force-writing** of the log
 - Ensures that redo operations can be done successfully

Concurrency

- Multiple transactions should be able to run concurrently
- Advantages
 - Increased processor and disk utilization leading to better *throughput*: one is using CPU, other is doing disk I/O
 - Reduced average response time: short transactions finish earlier and do not wait behind long ones
- **Concurrency control schemes** achieve isolation
- Must ensure **correctness** of concurrent executions
- **Serializability** imposes notion of correctness

Recovery management system

- **Recovery management system** of a database ensures that atomicity and durability properties are maintained despite failures
- **Fail-stop assumption**: Data on non-volatile storage is *not* lost due to system crash or power failure

Recovery management system

- **Recovery management system** of a database ensures that atomicity and durability properties are maintained despite failures
- **Fail-stop assumption**: Data on non-volatile storage is *not* lost due to system crash or power failure
- Recovery algorithms have two main parts
 - ① Action taken *during normal execution* to ensure that enough information is collected to recover from failures
 - ② Action taken *after a failure* that utilizes information collected so far to recover the database contents and maintain atomicity and durability
- Assume that transactions run serially, i.e., one after another

Log-Based Recovery

- **Log** is maintained on *stable storage*
 - Stable storage: data is *never* lost
- **Log records** for every operation of a transaction are recorded
- When transaction T starts, (start, T) record is logged
- *Before* read or write, the corresponding log record is written
- When T finishes successfully, (commit, T) is logged
- If T fails, (abort, T) is logged
- Two approaches of recovery based on logs
 - 1 **Deferred database modification**
 - 2 **Immediate database modification**
- Logs are written to stable storage

Deferred Database Modification

- Defers *all* writes to after partial commit
- Write log record is of the form (write, T , x , new)
 - Old value of x is not needed
- After T partially commits, (commit, T) is recorded in the log and log is written to the stable storage
 - (abort, T) record is not needed
- Log records are then read to actually perform the writes on the database

Deferred Database Modification

- Defers *all* writes to after partial commit
- Write log record is of the form (write, T , x , new)
 - Old value of x is not needed
- After T partially commits, (commit, T) is recorded in the log and log is written to the stable storage
 - (abort, T) record is not needed
- Log records are then read to actually perform the writes on the database
- When a transaction crashes, its (commit, T) record is searched
- If not found,

Deferred Database Modification

- Defers *all* writes to after partial commit
- Write log record is of the form (write, T , x , new)
 - Old value of x is not needed
- After T partially commits, (commit, T) is recorded in the log and log is written to the stable storage
 - (abort, T) record is not needed
- Log records are then read to actually perform the writes on the database
- When a transaction crashes, its (commit, T) record is searched
- If not found, nothing to do
 - No write has been performed on the database
- If found,

Deferred Database Modification

- Defers *all* writes to after partial commit
- Write log record is of the form (write, T, x, new)
 - Old value of x is not needed
- After T partially commits, (commit, T) is recorded in the log and log is written to the stable storage
 - (abort, T) record is not needed
- Log records are then read to actually perform the writes on the database
- When a transaction crashes, its (commit, T) record is searched
- If not found, nothing to do
 - No write has been performed on the database
- If found, redo all operations
 - Not sure if crash occurred while performing the deferred writes or after them
- Redo operation *must* be idempotent
 - Idempotent: Multiple operations has the same effect as single
- Also called a no-undo/redo recovery scheme
- Redos must be done in the order of serial transactions

Example

- (start, T0); (write, T0, A, 9); (commit, T0); (start, T1); (write, T1, A, 7); (commit, T1)
- Crash after (write, T0) statement

Example

- (start, T0); (write, T0, A, 9); (commit, T0); (start, T1); (write, T1, A, 7); (commit, T1)
- Crash after (write, T0) statement
 - Nothing needs to be done
- Crash after (write, T1) statement

Example

- (start, T0); (write, T0, A, 9); (commit, T0); (start, T1); (write, T1, A, 7); (commit, T1)
- Crash after (write, T0) statement
 - Nothing needs to be done
- Crash after (write, T1) statement
 - T0 is redone
- Crash after (commit, T1) statement

Example

- (start, T0); (write, T0, A, 9); (commit, T0); (start, T1); (write, T1, A, 7); (commit, T1)
- Crash after (write, T0) statement
 - Nothing needs to be done
- Crash after (write, T1) statement
 - T0 is redone
- Crash after (commit, T1) statement
 - Both T0 and T1 are redone
 - redo(T0) must be done before redo(T1)
 - Otherwise, value of A will be wrong

Immediate database modification

- Allows writes to modify database even for uncommitted transactions
- Write log record is of the form (write, T, x, old, new)
 - Old value of x is needed
- Log record of write must be written to stable storage before the corresponding write is performed on the database

Immediate database modification

- Allows writes to modify database even for uncommitted transactions
- Write log record is of the form (write, T, x, old, new)
 - Old value of x is needed
- Log record of write must be written to stable storage before the corresponding write is performed on the database
- When a transaction crashes, its (commit, T) record is searched
- If not found,

Immediate database modification

- Allows writes to modify database even for uncommitted transactions
- Write log record is of the form (write, T, x, old, new)
 - Old value of x is needed
- Log record of write must be written to stable storage before the corresponding write is performed on the database
- When a transaction crashes, its (commit, T) record is searched
- If not found, **undo** all operations
 - Uncommitted writes have been performed on the database
- If found,

Immediate database modification

- Allows writes to modify database even for uncommitted transactions
- Write log record is of the form (write, T, x, old, new)
 - Old value of x is needed
- Log record of write must be written to stable storage before the corresponding write is performed on the database
- When a transaction crashes, its (commit, T) record is searched
- If not found, **undo** all operations
 - Uncommitted writes have been performed on the database
- If found, **redo** all operations
 - If (commit, T) record is deferred till all writes are performed on the database,

Immediate database modification

- Allows writes to modify database even for uncommitted transactions
- Write log record is of the form (write, T, x, old, new)
 - Old value of x is needed
- Log record of write must be written to stable storage before the corresponding write is performed on the database
- When a transaction crashes, its (commit, T) record is searched
- If not found, **undo** all operations
 - Uncommitted writes have been performed on the database
- If found, **redo** all operations
 - If (commit, T) record is deferred till all writes are performed on the database, redo is not required
- Both undo and redo operations *must* be **idempotent**
 - Crash may occur multiple times
- Also called a **undo/redo** (or **undo/no-redo**) recovery scheme
- Undos must be done in the reverse order of serial transactions
- Redos must be done in the forward order of serial transactions
- All undos must be done before any redo

Example

- (start, T0); (write, T0, A, 3, 9); (commit, T0); (start, T1); (write, T1, B, 2, 5); (write, T1, A, 9, 7); (commit, T1);
- Crash after (write, T0, A) statement

Example

- (start, T0); (write, T0, A, 3, 9); (commit, T0); (start, T1); (write, T1, B, 2, 5); (write, T1, A, 9, 7); (commit, T1);
- Crash after (write, T0, A) statement
 - T0 is undone
 - Value of A is set back to 3
- Crash after (write, T1, B) statement

Example

- (start, T0); (write, T0, A, 3, 9); (commit, T0); (start, T1); (write, T1, B, 2, 5); (write, T1, A, 9, 7); (commit, T1);
- Crash after (write, T0, A) statement
 - T0 is undone
 - Value of A is set back to 3
- Crash after (write, T1, B) statement
 - T1 is undone
 - T0 is redone
 - undo(T1) must be done before redo(T0)
 - Value of A is restored to 9 and that of B to 2
- Crash after (commit, T1) statement

Example

- (start, T0); (write, T0, A, 3, 9); (commit, T0); (start, T1); (write, T1, B, 2, 5); (write, T1, A, 9, 7); (commit, T1);
- Crash after (write, T0, A) statement
 - T0 is undone
 - Value of A is set back to 3
- Crash after (write, T1, B) statement
 - T1 is undone
 - T0 is redone
 - undo(T1) must be done before redo(T0)
 - Value of A is restored to 9 and that of B to 2
- Crash after (commit, T1) statement
 - Both T0 and T1 are redone
 - redo(T0) must be done before redo(T1)
 - Value of A is set correctly to 7

Checkpoints

- Logs can become very big
- Searching in logs become time-consuming
- Transactions may be re-done unnecessarily
- **Checkpointing** alleviates these problems
- Log records are flushed to stable storage
- All pending writes are performed on the database
- An entry (checkpoint) is made in the log, and it is written to the stable storage

Checkpoints

- Logs can become very big
- Searching in logs become time-consuming
- Transactions may be re-done unnecessarily
- **Checkpointing** alleviates these problems
- Log records are flushed to stable storage
- All pending writes are performed on the database
- An entry (checkpoint) is made in the log, and it is written to the stable storage
- During recovery, any transaction that has completed successfully before the checkpoint, i.e., have a (commit, T) entry need not be considered
- Any T_i with (commit, T_i) after checkpoint is redone
- Any T_j with (start, T_j) but no (commit, T_j) is undone

Concurrent Transactions

- (checkpoint L) entry contains a list L of transactions active at that point
- After crash, scan backwards to last checkpoint
- Add T_i to redo-list when (commit, T_i) entry is found
- Add T_j to undo-list when (start, T_j) entry is found
- Add T_k in L to undo-list if not present in redo-list

Concurrent Transactions

- (checkpoint L) entry contains a list L of transactions active at that point
- After crash, scan backwards to last checkpoint
- Add T_i to redo-list when (commit, T_i) entry is found
- Add T_j to undo-list when (start, T_j) entry is found
- Add T_k in L to undo-list if not present in redo-list
- Undos done first in reverse order
- Redos done later in forward order
- Only operations after checkpoint need to be undone or redone

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list:

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4,

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list:

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order:

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order: T4,

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order: T4, T2,

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order: T4, T2, T3
- Order of operations:

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order: T4, T2, T3
- Order of operations:
 - T4:

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order: T4, T2, T3
- Order of operations:
 - T4: Revert value of *C* to 7
 - T2:

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order: T4, T2, T3
- Order of operations:
 - T4: Revert value of *C* to 7
 - T2: No operation
 - T3:

Example

- (start, T1); (write, T1, B, 2, 3); (start, T2); (commit, T1); (write, T2, C, 5, 7); (checkpoint, {T2}); (start, T3); (write, T3, A, 1, 9); (commit, T3); (start, T4); (write, T4, C, 7, 2)
- Undo-list: T4, T2
- Redo-list: T3
- Order: T4, T2, T3
- Order of operations:
 - T4: Revert value of *C* to 7
 - T2: No operation
 - T3: Re-write value of *A* to 9

Log Record Buffering

- Log records are buffered in memory before a block is output to the stable storage
- Records are flushed in the order of appearance in the log
- **Force-writing** is used to flush log records to stable storage before a transaction enters the commit point
- The (commit, T) entry is also flushed
- Before a block of data is written to the database, all log records pertaining to it are flushed to stable storage
- This rule is called **write-ahead logging** or **WAL** rule

Shadow Paging

- Maintain two page tables: **current page table** and **shadow page table**
- Shadow page table is maintained on disk without any change

Shadow Paging

- Maintain two page tables: **current page table** and **shadow page table**
- Shadow page table is maintained on disk without any change
- When a page requires change, it is copied to a new location
- Current page table points to the new location
- Updates are performed on the copy

Shadow Paging

- Maintain two page tables: **current page table** and **shadow page table**
- Shadow page table is maintained on disk without any change
- When a page requires change, it is copied to a new location
- Current page table points to the new location
- Updates are performed on the copy
- When a transaction commits, flush all changed pages to disk
- Modify shadow page table with contents of current page table

Shadow Paging

- Maintain two page tables: **current page table** and **shadow page table**
- Shadow page table is maintained on disk without any change
- When a page requires change, it is copied to a new location
- Current page table points to the new location
- Updates are performed on the copy
- When a transaction commits, flush all changed pages to disk
- Modify shadow page table with contents of current page table
- No recovery needed
- Also called a **no-undo/no-redo** recovery scheme

Shadow Paging

- Maintain two page tables: **current page table** and **shadow page table**
- Shadow page table is maintained on disk without any change
- When a page requires change, it is copied to a new location
- Current page table points to the new location
- Updates are performed on the copy
- When a transaction commits, flush all changed pages to disk
- Modify shadow page table with contents of current page table
- No recovery needed
- Also called a **no-undo/no-redo** recovery scheme
- Disadvantages

Shadow Paging

- Maintain two page tables: **current page table** and **shadow page table**
- Shadow page table is maintained on disk without any change
- When a page requires change, it is copied to a new location
- Current page table points to the new location
- Updates are performed on the copy
- When a transaction commits, flush all changed pages to disk
- Modify shadow page table with contents of current page table
- No recovery needed
- Also called a **no-undo/no-redo** recovery scheme
- Disadvantages
 - Entire page table is copied even though only some pages are modified
 - Commit overhead may be high
 - Works for serial transactions only
- Advantages

Shadow Paging

- Maintain two page tables: **current page table** and **shadow page table**
- Shadow page table is maintained on disk without any change
- When a page requires change, it is copied to a new location
- Current page table points to the new location
- Updates are performed on the copy
- When a transaction commits, flush all changed pages to disk
- Modify shadow page table with contents of current page table
- No recovery needed
- Also called a **no-undo/no-redo** recovery scheme
- Disadvantages
 - Entire page table is copied even though only some pages are modified
 - Commit overhead may be high
 - Works for serial transactions only
- Advantages
 - Recovery is built-in
 - No overhead of writing log records