CS315: Database Systems
Concurrency Control

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

Locks

- Locks are used to control access to a data item
- Lock requests are made to concurrency control manager
- Concurrency control manager decides whether and when to grant locks
- Locking and unlocking must be atomic

Locks

- Locks are used to control access to a data item
- Lock requests are made to concurrency control manager
- Concurrency control manager decides whether and when to grant locks
- Locking and unlocking must be atomic
- A data item can be locked in two modes
  1. Exclusive (X) mode: Data item can be both written and read
  2. Shared (S) mode: Data item can only be read

Locks

- Locks are used to control access to a data item
- Lock requests are made to concurrency control manager
- Concurrency control manager decides whether and when to grant locks
- Locking and unlocking must be atomic
- A data item can be locked in two modes
  1. Exclusive (X) mode: Data item can be both written and read
  2. Shared (S) mode: Data item can only be read
- A lock can be granted based on the compatibility matrix
- Lock compatibility matrix or conflict matrix

	S	X
S	yes	no
X	no	no

- If a lock cannot be granted, it must wait

Locking Protocol

- A schedule must specify all locking and unlocking operations and their modes
  - $lx(a)$ requests an exclusive lock on data item a; $ux(a)$ releases it
  - $ls(a)$ requests a shared lock on data item a; $us(a)$ releases it
- Example: $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); ux_1(a); us_2(b)$

Locking Protocol

- A schedule must specify all locking and unlocking operations and their modes
  - $lx(a)$ requests an exclusive lock on data item a; $ux(a)$ releases it
  - $ls(a)$ requests a shared lock on data item a; $us(a)$ releases it
- Example: $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); ux_1(a); us_2(b)$
- Consider $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); lx_2(a); lx_1(b)$

Locking Protocol

- A schedule must specify all locking and unlocking operations and their modes
  - $lx(a)$ requests an exclusive lock on data item a; $ux(a)$ releases it
  - $ls(a)$ requests a shared lock on data item a; $us(a)$ releases it
- Example: $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); ux_1(a); us_2(b)$
- Consider $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); lx_2(a); lx_1(b)$
  - Deadlock

Locking Protocol

- A schedule must specify all locking and unlocking operations and their modes
  - $lx(a)$ requests an exclusive lock on data item a; $ux(a)$ releases it
  - $ls(a)$ requests a shared lock on data item a; $us(a)$ releases it
- Example: $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); ux_1(a); us_2(b)$
- Consider $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); lx_2(a); lx_1(b)$
  - Deadlock
- Starvation may also happen

Locking Protocol

- A schedule must specify all locking and unlocking operations and their modes
  - $lx(a)$ requests an exclusive lock on data item a; $ux(a)$ releases it
  - $ls(a)$ requests a shared lock on data item a; $us(a)$ releases it
- Example: $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); ux_1(a); us_2(b)$
- Consider $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); lx_2(a); lx_1(b)$
  - Deadlock
- Starvation may also happen
- A locking protocol specifies the rules of how a transaction can acquire and release locks

Two-Phase Locking Protocol (2PL)

- Two phases
- Phase 1: Growing (locking) phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking (unlocking) phase
  - Transaction may release locks
  - Transaction may not obtain locks

Two-Phase Locking Protocol (2PL)

- Two phases
- Phase 1: Growing (locking) phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking (unlocking) phase
  - Transaction may release locks
  - Transaction may not obtain locks
- 2PL schedules are conflict serializable
  - Serialized in the order of lock points
  - Lock point is the time when *all* locks are obtained

Two-Phase Locking Protocol (2PL)

- Two phases
- Phase 1: Growing (locking) phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking (unlocking) phase
  - Transaction may release locks
  - Transaction may not obtain locks
- 2PL schedules are conflict serializable
  - Serialized in the order of lock points
  - Lock point is the time when *all* locks are obtained
- May suffer from deadlock
  - $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); ls_2(a); lx_1(b)$

Two-Phase Locking Protocol (2PL)

- Two phases
- Phase 1: Growing (locking) phase
  - Transaction may obtain locks
  - Transaction may not release locks
- Phase 2: Shrinking (unlocking) phase
  - Transaction may release locks
  - Transaction may not obtain locks
- 2PL schedules are conflict serializable
  - Serialized in the order of lock points
  - Lock point is the time when *all* locks are obtained
- May suffer from deadlock
  - $lx_1(a); r_1(a); w_1(a); ls_2(b); r_2(b); ls_2(a); lx_1(b)$
- May suffer from cascading rollbacks
  - $lx_1(a); r_1(a); w_1(a); ux_1(a); lx_2(a); r_2(a); w_2(a); ux_2(a); ls_3(a); r_3(a); a_1$

Variants of 2PL

- Basic 2PL
  - Basic protocol

Variants of 2PL

- Basic 2PL
  - Basic protocol
- Strict 2PL
  - A transaction must hold all its *exclusive* locks till it commits or aborts
  - Avoids cascading rollbacks
  - Produces strict schedules
  - May deadlock

Variants of 2PL

- Basic 2PL
  - Basic protocol
- Strict 2PL
  - A transaction must hold all its *exclusive* locks till it commits or aborts
  - Avoids cascading rollbacks
  - Produces strict schedules
  - May deadlock
- Rigorous 2PL
  - A transaction must hold *all* its locks till it commits or aborts
  - Transactions can be serialized in the order of their commits

Variants of 2PL

- Basic 2PL
  - Basic protocol
- Strict 2PL
  - A transaction must hold all its *exclusive* locks till it commits or aborts
  - Avoids cascading rollbacks
  - Produces strict schedules
  - May deadlock
- Rigorous 2PL
  - A transaction must hold *all* its locks till it commits or aborts
  - Transactions can be serialized in the order of their commits
- Conservative (static) 2PL
  - All locks are acquired atomically before a transaction begins
  - Each transaction declares its read set and write set
  - Deadlock-free

Timestamps

- Each transaction is assigned a timestamp when it starts
  - Transaction T_i starting earlier has a lower timestamp than T_j starting later
- For each data item x, two timestamps are maintained
- write-timestamp(x), wts(x), is the largest timestamp of any transaction that executed write successfully
- read-timestamp(x), rts(x), is the largest timestamp of any transaction that executed read successfully
- Protocols using timestamps *cannot* deadlock

Timestamp Ordering (TO) Protocol

- Ensures that conflicting operations are executed in timestamp order
- When a transaction T requests read(x)

Timestamp Ordering (TO) Protocol

- Ensures that conflicting operations are executed in timestamp order
- When a transaction T requests read(x)
  - If ts(T) $>$ wts(x), no conflict

Timestamp Ordering (TO) Protocol

- Ensures that conflicting operations are executed in timestamp order
- When a transaction T requests read(x)
  - If ts(T) $>$ wts(x), no conflict
    - read(x) request is executed
    - rts(x) is updated to max$\{$rts(x), ts(T)$\}$

Timestamp Ordering (TO) Protocol

- Ensures that conflicting operations are executed in timestamp order
- When a transaction T requests read(x)
  - If ts(T) $>$ wts(x), no conflict
    - read(x) request is executed
    - rts(x) is updated to max$\{$rts(x), ts(T)$\}$
  - If ts(T) $<$ wts(x), then T is attempting to read a value that has been overwritten by a later transaction

Timestamp Ordering (TO) Protocol

- Ensures that conflicting operations are executed in timestamp order
- When a transaction T requests read(x)
  - If ts(T) $>$ wts(x), no conflict
    - read(x) request is executed
    - rts(x) is updated to max$\{$rts(x), ts(T)$\}$
  - If ts(T) $<$ wts(x), then T is attempting to read a value that has been overwritten by a later transaction
    - read(x) request is rejected
    - T or transaction that produced wts(x) is rolled back

Timestamp Ordering (TO) Protocol (contd.)

- When a transaction T requests write(x)

Timestamp Ordering (TO) Protocol (contd.)

- When a transaction T requests write(x)
  - If ts(T) $>$ rts(x) and ts(T) $>$ wts(x), no conflict

Timestamp Ordering (TO) Protocol (contd.)

- When a transaction T requests write(x)
  - If ts(T) $>$ rts(x) and ts(T) $>$ wts(x), no conflict
    - write(x) request is executed
    - wts(x) is updated to ts(T)

Timestamp Ordering (TO) Protocol (contd.)

- When a transaction T requests write(x)
  - If ts(T) $>$ rts(x) and ts(T) $>$ wts(x), no conflict
    - write(x) request is executed
    - wts(x) is updated to ts(T)
  - If ts(T) $<$ rts(x), then value of x that is being written should have been read earlier

Timestamp Ordering (TO) Protocol (contd.)

- When a transaction T requests write(x)
  - If ts(T) $>$ rts(x) and ts(T) $>$ wts(x), no conflict
    - write(x) request is executed
    - wts(x) is updated to ts(T)
  - If ts(T) $<$ rts(x), then value of x that is being written should have been read earlier
    - write(x) request is rejected
    - T or transaction that produced rts(x) is rolled back

Timestamp Ordering (TO) Protocol (contd.)

- When a transaction T requests write(x)
  - If ts(T) $>$ rts(x) and ts(T) $>$ wts(x), no conflict
    - write(x) request is executed
    - wts(x) is updated to ts(T)
  - If ts(T) $<$ rts(x), then value of x that is being written should have been read earlier
    - write(x) request is rejected
    - T or transaction that produced rts(x) is rolled back
  - If ts(T) $<$ wts(x), then T is attempting to write an obsolete value that has been overwritten by a later transaction

Timestamp Ordering (TO) Protocol (contd.)

- When a transaction T requests write(x)
  - If ts(T) $>$ rts(x) and ts(T) $>$ wts(x), no conflict
    - write(x) request is executed
    - wts(x) is updated to ts(T)
  - If ts(T) $<$ rts(x), then value of x that is being written should have been read earlier
    - write(x) request is rejected
    - T or transaction that produced rts(x) is rolled back
  - If ts(T) $<$ wts(x), then T is attempting to write an obsolete value that has been overwritten by a later transaction
    - write(x) request is rejected
    - T or transaction producing wts(x) is rolled back

Correctness

- Guarantees conflict serializability
- Conflicting operations are executed in timestamp order
    - If an operation appears out of order, it is rejected

Correctness

- Guarantees conflict serializability
- Conflicting operations are executed in timestamp order
  - If an operation appears out of order, it is rejected
- No deadlock since all edges in the precedence graph are from transactions with smaller timestamp to those with larger timestamp

Correctness

- Guarantees conflict serializability
- Conflicting operations are executed in timestamp order
  - If an operation appears out of order, it is rejected
- No deadlock since all edges in the precedence graph are from transactions with smaller timestamp to those with larger timestamp
- May cause starvation

Correctness

- Guarantees conflict serializability
- Conflicting operations are executed in timestamp order
  - If an operation appears out of order, it is rejected
- No deadlock since all edges in the precedence graph are from transactions with smaller timestamp to those with larger timestamp
- May cause starvation
- Is not cascadeless

Correctness

- Guarantees conflict serializability
- Conflicting operations are executed in timestamp order
  - If an operation appears out of order, it is rejected
- No deadlock since all edges in the precedence graph are from transactions with smaller timestamp to those with larger timestamp
- May cause starvation
- Is not cascadeless
- Is not recoverable

Modifications

- To make it recoverable
  - Use commit dependency
  - If T_i reads from T_j and T_j has not committed, then T_i has a commit dependency on T_j
  - Ensure that T_i does not commit before T_j commits

Modifications

- To make it recoverable
  - Use commit dependency
  - If T_i reads from T_j and T_j has not committed, then T_i has a commit dependency on T_j
  - Ensure that T_i does not commit before T_j commits
- To make it recoverable and cascadeless
  - *All* writes are performed *atomically* in the end
  - A transaction, if aborts, is re-started with a new timestamp
- To make it recoverable and cascadeless
  - Lock data that is begin written
  - Wait for it to be committed before allowing read

Modifications

- To make it recoverable
  - Use commit dependency
  - If T_i reads from T_j and T_j has not committed, then T_i has a commit dependency on T_j
  - Ensure that T_i does not commit before T_j commits
- To make it recoverable and cascadeless
  - *All* writes are performed *atomically* in the end
  - A transaction, if aborts, is re-started with a new timestamp
- To make it recoverable and cascadeless
  - Lock data that is begin written
  - Wait for it to be committed before allowing read
- Strict timestamp ordering: to make it strict
  - Wait for data to be committed before reading or writing

Thomas' Write Rule

- Obsolete writes may be ignored
- When T attempts to write x, if ts(T) $<$ wts(x), then T is trying to write an obsolete value of x
- Thomas' write rule: Rather than aborting T, ignore the write operation
    - Write is obsolete anyway

Thomas' Write Rule

- Obsolete writes may be ignored
- When T attempts to write x, if ts(T) < wts(x), then T is trying to write an obsolete value of x
- Thomas' write rule: Rather than aborting T, ignore the write operation
  - Write is obsolete anyway
- Improves concurrency and recoverability
- Allows some view-serializable schedules that are not conflict-serializable
  - $r_1(a) w_2(a) w_1(a) w_3(a)$

Validation (Certification)-Based Protocol

- Three phases of a transaction T
- Read and execution phase: T writes only to local temporary variables
- Validation phase: T performs validation test to determine if local variables can be written without violating serializability
- Write phase: If T is validated, it updates the database; otherwise it is rolled back (actually nothing needs to be done)
- Also known as optimistic concurrency control since transaction executes fully in the hope that all is well

Validation (Certification)-Based Protocol

- Three phases of a transaction T
- Read and execution phase: T writes only to local temporary variables
- Validation phase: T performs validation test to determine if local variables can be written without violating serializability
- Write phase: If T is validated, it updates the database; otherwise it is rolled back (actually nothing needs to be done)
- Also known as optimistic concurrency control since transaction executes fully in the hope that all is well
- Three timestamps for each transaction
  - *start(T)*: start of execution phase
  - *validation(T)*: start of validation phase
  - *finish(T)*: end of write phase
- Timestamp of T is set to validation timestamp: ts(T) = validation(T)
- Serialized using this timestamp
  - Increases concurrency

Validation (Certification)-Based Protocol

- Three phases of a transaction T
- Read and execution phase: T writes only to local temporary variables
- Validation phase: T performs validation test to determine if local variables can be written without violating serializability
- Write phase: If T is validated, it updates the database; otherwise it is rolled back (actually nothing needs to be done)
- Also known as optimistic concurrency control since transaction executes fully in the hope that all is well
- Three timestamps for each transaction
  - *start(T)*: start of execution phase
  - *validation(T)*: start of validation phase
  - *finish(T)*: end of write phase
- Timestamp of T is set to validation timestamp: ts(T) = validation(T)
- Serialized using this timestamp
  - Increases concurrency
- Cascadeless

Validation (Certification)-Based Protocol

- Three phases of a transaction T
- Read and execution phase: T writes only to local temporary variables
- Validation phase: T performs validation test to determine if local variables can be written without violating serializability
- Write phase: If T is validated, it updates the database; otherwise it is rolled back (actually nothing needs to be done)
- Also known as optimistic concurrency control since transaction executes fully in the hope that all is well
- Three timestamps for each transaction
  - *start(T)*: start of execution phase
  - *validation(T)*: start of validation phase
  - *finish(T)*: end of write phase
- Timestamp of T is set to validation timestamp: ts(T) = validation(T)
- Serialized using this timestamp
  - Increases concurrency
- Cascadeless
- Starvation

Validation (Certification)-Based Protocol

- Three phases of a transaction T
- Read and execution phase: T writes only to local temporary variables
- Validation phase: T performs validation test to determine if local variables can be written without violating serializability
- Write phase: If T is validated, it updates the database; otherwise it is rolled back (actually nothing needs to be done)
- Also known as optimistic concurrency control since transaction executes fully in the hope that all is well
- Three timestamps for each transaction
  - *start(T)*: start of execution phase
  - *validation(T)*: start of validation phase
  - *finish(T)*: end of write phase
- Timestamp of T is set to validation timestamp: ts(T) = validation(T)
- Serialized using this timestamp
  - Increases concurrency
- Cascadeless
- Starvation

Validation Test

- For a transaction T_i, check two conditions for all transactions T_j with $\text{ts}(T_j) < \text{ts}(T_i)$
  - $\text{finish}(T_j) < \text{start}(T_i)$
  - $\text{finish}(T_j) < \text{validation}(T_i)$ and the read-set of T_i is disjoint from the write-set of T_j
- If either of these conditions is true, validation succeeds; otherwise, it fails

Validation Test

- For a transaction T_i, check two conditions for all transactions T_j with $\text{ts}(T_j) < \text{ts}(T_i)$
  - $\text{finish}(T_j) < \text{start}(T_i)$
  - $\text{finish}(T_j) < \text{validation}(T_i)$ and the read-set of T_i is disjoint from the write-set of T_j
- If either of these conditions is true, validation succeeds; otherwise, it fails
- Justification
  - First condition ensures serial schedules
  - Writes of T_i cannot affect reads of T_j

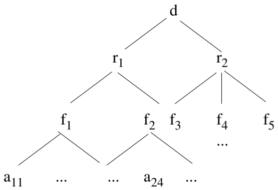
Validation Test

- For a transaction T_i, check two conditions for all transactions T_j with $\text{ts}(T_j) < \text{ts}(T_i)$
  - $\text{finish}(T_j) < \text{start}(T_i)$
  - $\text{finish}(T_j) < \text{validation}(T_i)$ and the read-set of T_i is disjoint from the write-set of T_j
- If either of these conditions is true, validation succeeds; otherwise, it fails
- Justification
  - First condition ensures serial schedules
  - Writes of T_i cannot affect reads of T_j
  - Writes of T_j do not affect reads of T_i as they are disjoint

Mutliple Granularity

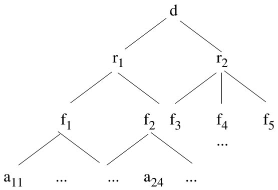
- Hierarchy of data items
  - DB, Relation, Tuple, Attribute
- Locking can be done at different levels
- Locking a node *explicitly* locks all its descendants *implicitly*
  - Explicit locks
  - Implicit locks
- Granularity of locking
  - *Fine granularity*: lower in tree, high concurrency, high locking overhead
  - *Coarse granularity*: higher in tree, low concurrency, low locking overhead

Problems of Simple Locking



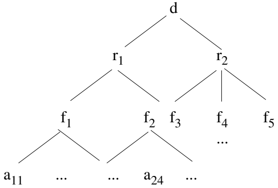
- Assume that T_1 has locked tuple f_2
- T_2 wants to lock a_{24}

Problems of Simple Locking



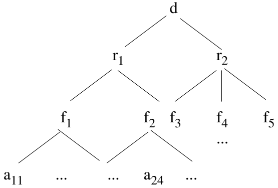
- Assume that T_1 has locked tuple f_2
- T_2 wants to lock a_{24}
  - It cannot since a_{24} is implicitly locked

Problems of Simple Locking



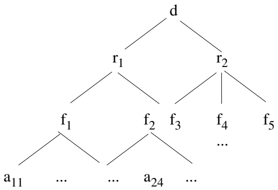
- Assume that T_1 has locked tuple f_2
- T_2 wants to lock a_{24}
  - It cannot since a_{24} is implicitly locked
  - Find out by traversing path from a_{24} to d
- T_3 wants to lock r_1

Problems of Simple Locking



- Assume that T_1 has locked tuple f_2
- T_2 wants to lock a_{24}
  - It cannot since a_{24} is implicitly locked
  - Find out by traversing path from a_{24} to d
- T_3 wants to lock r_1
  - It cannot since that would lock f_2 implicitly

Problems of Simple Locking



- Assume that T_1 has locked tuple f_2
- T_2 wants to lock a_{24}
  - It cannot since a_{24} is implicitly locked
  - Find out by traversing path from a_{24} to d
- T_3 wants to lock r_1
  - It cannot since that would lock f_2 implicitly
  - Find out by searching entire subtree under r_1
- Thus, for efficiency, intention lock modes are used
  - Ancestors of an explicitly locked node are in intention mode

Intention Lock Modes

- In addition to shared (S) and exclusive (X) locks, three additional locks
- Intention-shared (IS): at least one descendant has a S lock
- Intention-exclusive (IX): at least one descendant has a X lock
- Shared and intention-exclusive (SIX): node is locked in S mode *and* at least one descendant has X lock

Rules of Locking

- A transaction may obtain only one lock on an entity at a time
- If two locks are needed, the more restrictive one will be acquired
- Every lock must be given notice of all lower-level locks

Rules of Locking

- A transaction may obtain only one lock on an entity at a time
- If two locks are needed, the more restrictive one will be acquired
- Every lock must be given notice of all lower-level locks
- Locks are acquired in root-to-leaf order
- Locks are released in leaf-to-root order

Rules of Locking

- A transaction may obtain only one lock on an entity at a time
- If two locks are needed, the more restrictive one will be acquired
- Every lock must be given notice of all lower-level locks
- Locks are acquired in root-to-leaf order
- Locks are released in leaf-to-root order
- Compatibility matrix

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

Multiple Granularity Locking Scheme

- Transaction T wants to lock a node *x*:
  - Lock compatibility matrix is observed
  - In S or IS mode: only if parent of *x* is locked by T in IX or IS mode
  - In X, SIX or IX mode: only if parent of *x* is locked by T in IX or SIX mode
  - Maintains 2PL, i.e., has not unlocked anything

Multiple Granularity Locking Scheme

- Transaction T wants to lock a node x:
  - Lock compatibility matrix is observed
  - In S or IS mode: only if parent of x is locked by T in IX or IS mode
  - In X, SIX or IX mode: only if parent of x is locked by T in IX or SIX mode
  - Maintains 2PL, i.e., has not unlocked anything
- Transaction T wants to unlock a node x:
  - No child of x is currently locked by T

Multiple Granularity Locking Scheme

- Transaction T wants to lock a node *x*:
  - Lock compatibility matrix is observed
  - In S or IS mode: only if parent of *x* is locked by T in IX or IS mode
  - In X, SIX or IX mode: only if parent of *x* is locked by T in IX or SIX mode
  - Maintains 2PL, i.e., has not unlocked anything
- Transaction T wants to unlock a node *x*:
  - No child of *x* is currently locked by T
- Ensures conflict serializability

SIX Lock

- Suppose T_1 wants to read r_1 but only modify a_{24}
- Locking r_1 in IX mode will allow other transactions to lock r_1 in IX mode
  - Unsafe as T_1 is reading r_1
- Locking r_1 in S mode will allow other transactions to lock r_1 in S mode and read everything
  - Unsafe as T_1 is modifying a_{24}
- SIX lock compromises and is safer

Example

- T_1 wants to read a_{12}

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1
  - Locks d, r_1 in IS mode and f_1 in S mode
- T_4 wants to read d

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1
  - Locks d, r_1 in IS mode and f_1 in S mode
- T_4 wants to read d
  - Locks d in S mode

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1
  - Locks d, r_1 in IS mode and f_1 in S mode
- T_4 wants to read d
  - Locks d in S mode
- T_1 and T_2

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1
  - Locks d, r_1 in IS mode and f_1 in S mode
- T_4 wants to read d
  - Locks d in S mode
- T_1 and T_2 can execute concurrently
- T_1, T_3 and T_4

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1
  - Locks d, r_1 in IS mode and f_1 in S mode
- T_4 wants to read d
  - Locks d in S mode
- T_1 and T_2 can execute concurrently
- T_1, T_3 and T_4 can execute concurrently
- T_2 and T_3

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1
  - Locks d, r_1 in IS mode and f_1 in S mode
- T_4 wants to read d
  - Locks d in S mode
- T_1 and T_2 can execute concurrently
- T_1, T_3 and T_4 can execute concurrently
- T_2 and T_3 cannot execute concurrently
- T_2 and T_4

Example

- T_1 wants to read a_{12}
  - Locks d, r_1, f_1 in IS mode and a_{12} in S mode
- T_2 wants to write a_{14}
  - Locks d, r_1, f_1 in IX mode and a_{12} in X mode
- T_3 wants to read f_1
  - Locks d, r_1 in IS mode and f_1 in S mode
- T_4 wants to read d
  - Locks d in S mode
- T_1 and T_2 can execute concurrently
- T_1, T_3 and T_4 can execute concurrently
- T_2 and T_3 cannot execute concurrently
- T_2 and T_4 cannot execute concurrently

Deadlock Prevention

- Deadlock prevention schemes *never* allow a system to enter deadlock
- Two schemes that use timestamps

Deadlock Prevention

- Deadlock prevention schemes *never* allow a system to enter deadlock
- Two schemes that use timestamps
- Wait-die: Non-preemptive
  - Older transactions wait for younger ones to release data item (wait)
  - Younger ones do not wait, they roll back (die)
  - Young transactions may die many times

Deadlock Prevention

- Deadlock prevention schemes *never* allow a system to enter deadlock
- Two schemes that use timestamps
- Wait-die: Non-preemptive
  - Older transactions wait for younger ones to release data item (wait)
  - Younger ones do not wait, they roll back (die)
  - Young transactions may die many times
- Wound-wait: Preemptive
  - Older transactions kill younger ones and force them to release data item (wound)
  - Younger ones wait (wait)

Deadlock Prevention

- Deadlock prevention schemes *never* allow a system to enter deadlock
- Two schemes that use timestamps
- Wait-die: Non-preemptive
  - Older transactions wait for younger ones to release data item (wait)
  - Younger ones do not wait, they roll back (die)
  - Young transactions may die many times
- Wound-wait: Preemptive
  - Older transactions kill younger ones and force them to release data item (wound)
  - Younger ones wait (wait)
- Transactions are re-started with the *same* timestamp

Deadlock Prevention

- Deadlock prevention schemes *never* allow a system to enter deadlock
- Two schemes that use timestamps
- Wait-die: Non-preemptive
  - Older transactions wait for younger ones to release data item (wait)
  - Younger ones do not wait, they roll back (die)
  - Young transactions may die many times
- Wound-wait: Preemptive
  - Older transactions kill younger ones and force them to release data item (wound)
  - Younger ones wait (wait)
- Transactions are re-started with the *same* timestamp
- No starvation

Deadlock Prevention

- Deadlock prevention schemes *never* allow a system to enter deadlock
- Two schemes that use timestamps
- Wait-die: Non-preemptive
  - Older transactions wait for younger ones to release data item (wait)
  - Younger ones do not wait, they roll back (die)
  - Young transactions may die many times
- Wound-wait: Preemptive
  - Older transactions kill younger ones and force them to release data item (wound)
  - Younger ones wait (wait)
- Transactions are re-started with the *same* timestamp
- No starvation
- Wound-wait has fewer rollbacks than wait-die
  - Less likely for old transactions to not finish and want a lock from a young transaction

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?
  - One with lowest cost

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?
  - One with lowest cost
  - One with least progress

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?
  - One with lowest cost
  - One with least progress
  - One inducing least number of cascading rollbacks

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?
  - One with lowest cost
  - One with least progress
  - One inducing least number of cascading rollbacks
- How far to rollback?

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?
  - One with lowest cost
  - One with least progress
  - One inducing least number of cascading rollbacks
- How far to rollback?
  - Total rollback: Completely abort and re-start
  - Partial rollback: Rollback to only as far as necessary to break deadlock

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?
  - One with lowest cost
  - One with least progress
  - One inducing least number of cascading rollbacks
- How far to rollback?
  - Total rollback: Completely abort and re-start
  - Partial rollback: Rollback to only as far as necessary to break deadlock
- Starvation happens when the same transaction is repeatedly chosen as the victim

Deadlock Recovery

- Deadlocks can be detected by a wait-for graph
  - T_i waits for T_j, T_j waits for T_k, etc.
- If deadlock is detected by finding a cycle, a transaction must be chosen for roll back, i.e., made a victim
- Which transaction?
  - One with lowest cost
  - One with least progress
  - One inducing least number of cascading rollbacks
- How far to rollback?
  - Total rollback: Completely abort and re-start
  - Partial rollback: Rollback to only as far as necessary to break deadlock
- Starvation happens when the same transaction is repeatedly chosen as the victim
  - Factor number of rollbacks when choosing victim

Insert and Delete

- insert(x): inserts the data item x
- delete(x): deletes the data item x
- Logical errors
  - read(x), write(x) before insert(x)
  - read(x), write(x) after delete(x)
  - delete(x) after delete(x)
  - insert(x) after insert(x)
- Conflicts
  - Similar to write(x)
  - Conflicts with operations on relation

Phantom Phenomenon

- Transaction T1 reads an entire relation to compute an aggregate
- Transaction T2 inserts tuples to the relation
- Transactions T1 and T2 *logically conflict*

Phantom Phenomenon

- Transaction T1 reads an entire relation to compute an aggregate
- Transaction T2 inserts tuples to the relation
- Transactions T1 and T2 *logically conflict*
- However, they do not conflict on any tuple
- They conflict on a phantom tuple
  - This is called phantom phenomenon

Phantom Phenomenon

- Transaction T1 reads an entire relation to compute an aggregate
- Transaction T2 inserts tuples to the relation
- Transactions T1 and T2 *logically conflict*
- However, they do not conflict on any tuple
- They conflict on a phantom tuple
  - This is called phantom phenomenon
- Exclusive lock on relation solves this
- Multi-granularity locking

Phantom Phenomenon

- Transaction T1 reads an entire relation to compute an aggregate
- Transaction T2 inserts tuples to the relation
- Transactions T1 and T2 *logically conflict*
- However, they do not conflict on any tuple
- They conflict on a phantom tuple
  - This is called phantom phenomenon
- Exclusive lock on relation solves this
- Multi-granularity locking
- If index structure is used, index locking protocol improves concurrency by locking index nodes
  - Avoids phantom phenomenon since every transaction needs to lock all accessed nodes