ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Διάλεξη 9η: Transactions - part 2

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

Transaction Management

- Comparison of Undo and Redo Logging:
  - Undo logging: data must be written to disk immediately after a transaction finishes; potentially increases the number of I/O operations required
  - Redo logging: modified blocks must be kept in buffers until the transaction commits and log has been flushed; potentially increases the number of buffers required by transactions
  - Both may impose contradictory requirements during checkpointing
- Undo/Redo logging is more flexible than Undo or Redo logging
  - … but is also more costly

Undo/Redo Logging

- Undo/Redo log records differ only in the update records: <T,x,v,w> means that transaction T changed the value of DB element X from v to w
- Rule: before modifying any DB element X on disk, the update record <T,x,v,w> must appear on disk
  - <COMMIT T> record may precede or follow any of the changes to the DB elements on disk
  - more liberal than Undo or Redo logging
- For recovery, it permits either restoring the DB state or repeating the changes made. Policy:
  - Redo committed transactions (earliest first)
  - Undo incomplete transactions (latest first)
  - Both are necessary to avoid incomplete recovery

Recovery with Undo/Redo Logging

- Example:

	<START T>
Read (A,t); t ← t×2	
Write (A,t);	<T,A,8,16>
Read (B,t); t ← t×2	
Write (B,t);	<T,B,8,16>
FLUSH LOG;	
Output (A);	<COMMIT T>
Output (B);	

If failure occurs after <COMMIT T> has been written on disk, T is treated as committed and the value 16 is written on disk for A and B

If failure occurs before <COMMIT T> appears on disk, T is treated as an incomplete transaction and the value 8 is written on disk for A and B

Recovery with Undo/Redo Logging

- An additional rule may be used in order to avoid situations where, due to delayed commitment, a transaction appears (to the user) to have committed, but the <COMMIT > record has not been written on disk
  - a crash would cause such a transaction to be undone
  - Rule: A <COMMIT T> record must be flushed to disk as soon as it appears in the log
- Order of redo and undo during recovery:
  - doesn’t really matter
  - we still cannot prevent a committed transaction that must be redone to read a value written by an incomplete transaction that must be undone (dirty read)
  - need to isolate such transactions (concurrency control)

Checkpointing with Undo/Redo Logging

- Nonquiescent checkpointing:
  1. Write <START CKPT(T1,T2,…,Tk)> to log for active transactions T1,T2,…,Tk and flush the log
  2. Write to disk all dirty buffers (those that contain changed DB elements)
  3. Write <END CKPT> and flush the log
- Step 2 enforces the writing on disk of elements changed by incomplete transactions
- The following constraint must be observed though:
  - A transaction may not write any values until it is certain not to abort

Checkpointing with Undo/Redo Logging

- Example:

<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT(T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

During the checkpoint, A and B will be flushed to disk.

if failure occurs at the end, T2 and T3 are considered complete and are redone; T1 is considered to be complete and to have had its changes written on disk

if failure occurs before <COMMIT T3>, then T2 is considered complete whereas T3 is incomplete; redo T2 by writing 15 for C on disk; no need to write 10 for B; undo T3 by writing 19 for D; if T3 were active at the start of the checkpoint, we would need to check if other changes by T3 need to be undone

Concurrency Control

- I.e., how to maintain the DB in a consistent state in the presence of constraints when multiple transactions execute concurrently
- Example:

T1:	Read(A)	T2:	Read(A)
	$A \leftarrow A+100$		$A \leftarrow A\times2$
	Write(A)		Write(A)
	Read(B)		Read(B)
	$B \leftarrow B+100$		$B \leftarrow B\times2$
	Write(B)		Write(B)

Constraint: $A=B$

Schedules

Schedule A

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedules

Schedule B

T1	T2	A	B
		25	25
	Read(A);A ← A×2;		
	Write(A);	50	
	Read(B);B ← B×2;		
	Write(B);		50
Read(A); A ← A+100			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);			150
		150	150

Schedules

Schedule C

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
Read(B); B ← B+100;			
Write(B);			125
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedules

Schedule D

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		50
Read(B);B ← B+100;			
Write(B);			150
		250	150

Schedules

Schedule E

T1	T2’	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×1;		
	Write(A);	125	
	Read(B);B ← B×1;		
	Write(B);		25
Read(B);B ← B+100;			
Write(B);			125
		125	125

Transaction Scheduling

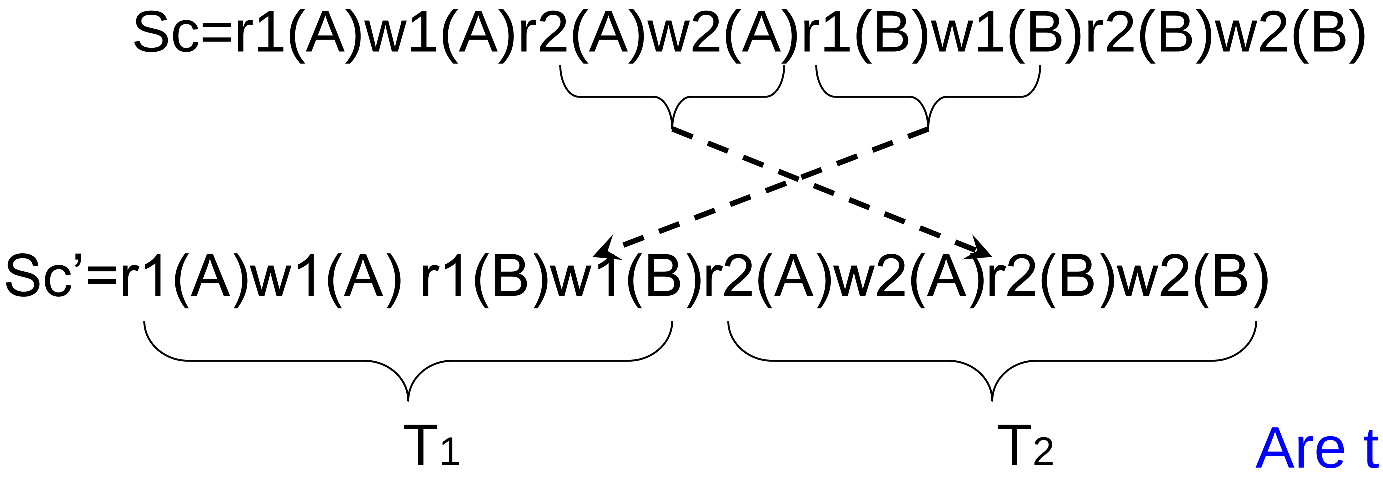
- Only want to allow the execution of “good” schedules regardless of
  - initial states
  - transaction semantics
- Only the order of reads and writes matters
- Example: given a schedule of the form

$$Sc=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

determine whether it should be allowed to proceed without causing inconsistency problems

Transaction Scheduling

- Scheduler must chose a “correct” interleaving of transaction operations among all possible interleavings
- Recall that transactions should appear to execute in isolation
- Hence, a “correct” interleaving should behave as if the transactions involved in the schedule were executed in isolation
- Example:



Sc=r1(A)w1(A)r2(A)w2(A)r1(B)w1(B)r2(B)w2(B)

Sc’=r1(A)w1(A) r1(B)w1(B)r2(A)w2(A)r2(B)w2(B)

Are the two equivalent?

Transaction Scheduling

- Example: Which is the equivalent serial schedule of:

  Sd=r1(A)w1(A)r2(A)w2(A) r2(B)w2(B)r1(B)w1(B)

T1 must precede T2 in any equivalent schedule

T2 must precede T1 in any equivalent schedule

Sd cannot be rearranged into a serial schedule, hence Sd is not equivalent to any serial schedule, hence Sd is “bad”

But, $Sc=r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$ is good

Transaction Concepts

- *Transaction:* sequence of ri(x), wi(x) actions
- *Conflicting actions:* r1(A) w2(A) w1(A)
  w2(A) r1(A) w2(A)
- *Schedule:* represents chronological order in which actions are executed
- *Serial schedule:* no interleaving of transactions
- What about concurrent schedules?
  - if they don’t involve conflicting actions, then any low-level synchronization mechanism is sufficient for scheduling

Definitions

- Schedules S1, S2 are conflict equivalent if S1 can be transformed into S2 by a series of swaps on non-conflicting actions.
- A schedule is conflict serializable if it is conflict equivalent to some serial schedule.
- For a schedule S, its precedence graph P(S) is defined as follows:
  - Nodes: transactions in S
  - Arcs: Ti → Tj whenever
    - pi(A), qj(A) are actions in S
    - $pi(A) <_S qj(A)$
    - at least one of pi, qj is a write

Results

- Lemma S1, S2 conflict equivalent ⇒ P(S1)=P(S2)
- Proof: Assume P(S1) ≠ P(S2)

  Then, ∃ Ti: Ti → Tj in S1 and not in S2

  S1 = …pi(A)... qj(A)… pi, qj

  S2 = …qj(A)…pi(A)... conflict

  Hence, S1 and S2 cannot be conflict equivalent
- Note: P(S1)=P(S2) does not imply S1, S2 conflict equivalent
- Counterexample:

$S_1=w_1(A)\ r_2(A) \quad w_2(B)\ r_1(B)$

$S_2=r_2(A)\ w_1(A) \quad r_1(B)\ w_2(B)$

Results

- Theorem: P(S1) acyclic iff S1 conflict serializable
- Proof:
  - (if) Assume S1 is conflict serializable

    ⇒ ∃ Ss: Ss, S1 conflict equivalent

    ⇒ P(Ss) = P(S1)

    ⇒ P(S1) acyclic since P(Ss) is acyclic
  - (only if) Assume P(S_1) is acyclic

    Transform S_1 as follows:

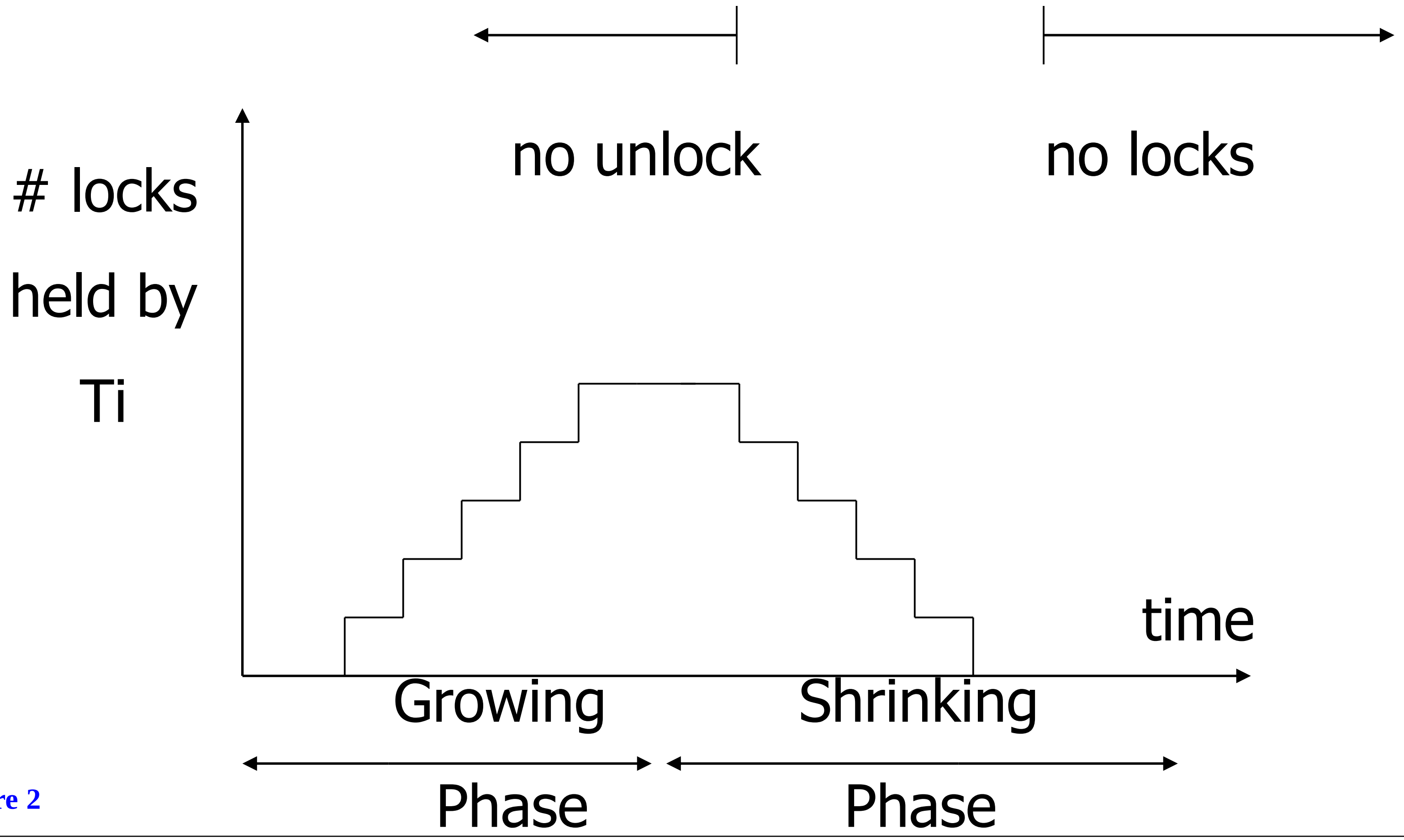
    1. Take T1 to be transaction with no incoming arcs
    2. Move all T1 actions to the front
    3. We now have S1 = < T1 actions ><... rest ...>; remove T1 and incident arcs
    4. Repeat above steps to serialize rest!

Enforcing Serializability

- Option 1: allow any schedule; check for cycles in precedence graph (reactive)
- Option 2: prevent cycles in precedence graphs of schedules (proactive)
- Locking protocols are used to implement Option 2
  - New actions: lock (exclusive), unlock
  - Scheduler maintains information about locks in a lock table
- Rule 1: well-formed transactions
  - A transaction is well-formed when every operation on a database item X is preceded by a lock request on X and followed by an unlock request on X
- Rule 2: legal schedule
  - A schedule is legal if no lock request is granted to a transaction Tj for a database item X when a transaction Ti has already been granted the lock to X

Concurrency Control

- 2-Phase Locking: all lock requests precede all unlock requests
  - Idea: T_i = ……. $l_i(A)$ ………... $u_i(A)$ ……...



no unlock

no locks

locks held by Ti

time

Growing Phase

Shrinking Phase

2-Phase Locking

- Intuitively, each 2PL transaction may be thought to execute in its entirety at the moment it releases the first item
- The conflict-equivalent serial schedule for a schedule S of 2PL transactions is the one in which transactions are ordered in the same order as their first unlocks
- Conversion: by induction on the number n of transactions in a legal schedule S
  - Note: conversion requires swapping the order of read and write operations of different transactions; while that is done, locks and unlocks can be ignored; once the actions are arranged serially, lock / unlock operations can be added

2-Phase Locking

- Conversion by Induction
  - Base case: n=1; S is already a serial schedule
  - Induction:

Let S involve transactions T1, T2, …, Tn; let Ti be the first transaction to unlock an item by op. Ui(X). Then, it is possible to move all read/write actions of Ti to the beginning of S without passing any conflicting action.

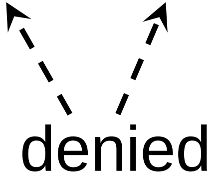
Let Ti include an action Wi(Y). If there existed an action Wj(Y) in S that precedes Wi(Y), then Uj(Y) and Li(Y) must also appear between Wj(Y) and Wi(Y).

We assumed that Ui(X) is the first unlock, hence it precedes Uj(Y). This means that Ui(X) must also appear before Li(Y). But then, Ti is not a 2PL transaction.

2PL and Deadlocks

- 2PL cannot prevent deadlocks
- Example: consider the following schedule :

L1(A), R1(A), L2(B), R2(B), W1(A), W2(B), L1(B), L2(A)


denied

Each transaction waits for the other to release a lock

Beyond Simple 2PL

- Improvements to 2PL’s performance for allowing more concurrency:
  - shared locks
  - multiple granularities
  - inserts, deletes and phantoms
  - other types of concurrency control mechanisms
- Shared Locks
  - so far, locks have been of a single type: exclusive
  - read operations do not conflict
  - no need to lock exclusively for read
  - SLi(X): “Ti requests a shared lock on X”
  - XLi(X): “Ti requests an exclusive lock on X”
  - Ui(X): “Ti releases lock on X”

Requirements

- Consistency
  - An action Ri(X) must be preceded by SLi(X) or XLi(X) with no intervening Ui(X)
  - An action Wi(X) must be preceded by XLi(X) with no intervening Ui(X)
  - All locks must be followed by an unlock of the same element
- 2PL
  - For any 2PL transaction Ti, no SLi(X) or XLi(X) can be preceded by Ui(X)
- Legality
  - If XLi(X) appears in a schedule, then there cannot be a following XLj(X) or SLj(X) for j<>i without an intervening Ui(X)
  - If SLi(X) appears in a schedule, then there cannot be a following XLj(X) for j<>i without an intervening Ui(X)

Example

- Consider the following 2PL transactions

  T1: SL1(A), R1(A), XL1(B), R1(B), W1(B), U1(A), U1(B)

  T2: SL2(A), R2(A), SL2(B), R2(B), U2(A), U2(B)

- A legal interleaved execution of T1, T2 is as follows:

T1	T2
SL1(A), R1(A)	
	SL2(A), R2(A), SL2(B), R2(B)
XL1(B) wait	
	U2(A), U2(B)
XL1(B), R1(B), W1(B)	
U1(A), U1(B)	

Upgrading Locks

- A transaction that wants to read and write an item, may first obtain a shared lock on the item for reading and then upgrade it to an exclusive lock for writing
- Example:

T1: SL1(A), R1(A), SL1(B), R1(B), XL1(B), W1(B), U1(A), U1(B)

T2: SL2(A), R2(A), SL2(B), R2(B), U2(A), U2(B)

T1	T2
SL1(A), R1(A)	
	SL2(A), R2(A), SL2(B), R2(B)
SL1(B), R1(B),	
XL1(B) wait	
	U2(A), U2(B)
XL1(B), W1(B), U1(A), U1(B)	

Upgrading Locks

- Lock upgrading may lead to deadlocks
- Example:

T1	T2
SL1(A), R1(A)	
	SL2(A), R2(A)
XL1(A) wait	
	XL2(A) wait

Update Locks

- We can avoid deadlocks caused by lock upgrade by using update locks:
    - An update lock ULi(X) allows transaction Ti to read X but not to write X
    - Only an update lock can be upgraded to a write lock
    - An update lock can be granted on X when there are already shared locks on X
    - Once there is an update lock on X , no additional locks are allowed on X (otherwise such a lock would never be upgraded to exclusive)

Example

T1: UL1(A), R1(A), XL1(A), W1(A), U1(A)

T2: UL2(A), R2(A), XL2(A), W2(A), U2(A)

T1	T2
UL1(A), R1(A)	
	UL2(A) wait
XL1(A), W1(A), U1(A)	
	UL2(A), R2(A), XL2(A), W2(A), U2(A)

Increment Locks

- Several transactions operate on DB items by simply adding or subtracting constants
    - E.g., money transfer between accounts, seat reservations
- Such transactions commute with each other and their relative order doesn’t matter
- However, they don’t commute with transactions that read or write
- Assume transactions may include operations of the form INC(A,c), meaning that constant c is to be added to DB element A
- INC(A,c) stands for: Read(A,t); t:=t+c; Write(A,t);
- Increment actions need increment locks: ILi(X)

Increment Locks

- Increment locks do not enable reads or writes
- Requirements:
  - A consistent transaction can perform INCi(X) only if it is preceded by ILi(X)
  - In a legal schedule, any number of transactions can hold an increment lock on item X at a time. If a transaction has an increment lock on X, no other transaction can have a shared or exclusive lock on X at the same time.
  - INCi(X) conflicts with Rj(X) and Wj(X) for j<>i
  - INCi(X) does not conflict with INCj(X)

Example

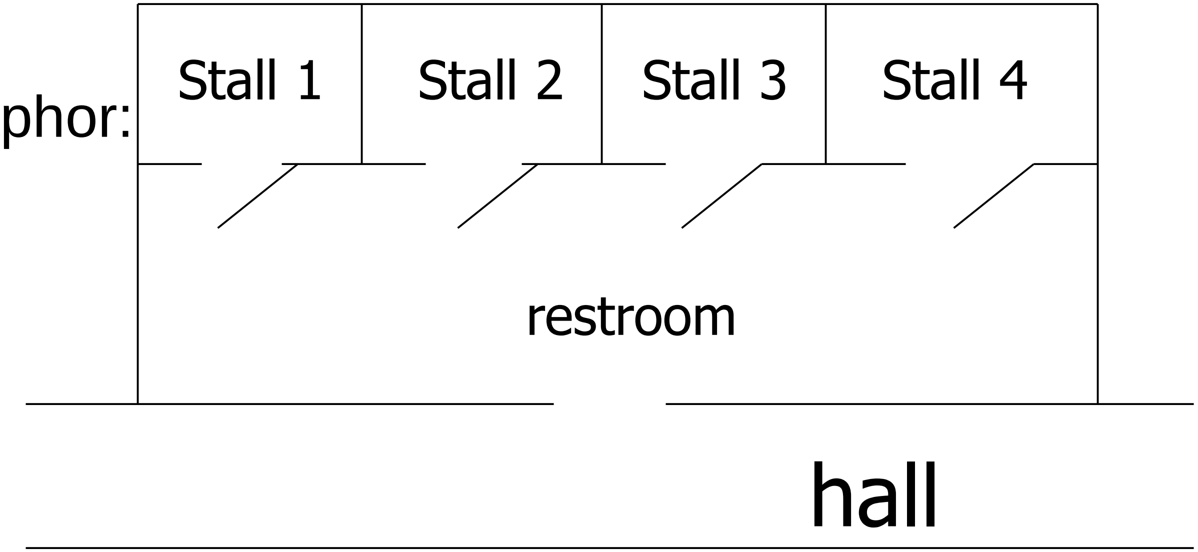
T1: SL1(A), R1(A), IL1(B), INC1(B), U1(A), U1(B)

T2: SL2(A), R2(A), IL2(B), INC2(B), U2(A), U2(B)

T1	T2
SL1(A), R1(A)	
	SL2(A), R2(A),IL2(B), INC2(B)
IL1(B), INC1(B)	
	U2(A), U2(B)
U1(A), U1(B)	

Granularity Issues

- Locking works well, but should we lock small or large objects?
- If large objects (e.g., relations) are locked
  - Need few locks
  - Low concurrency
- If small objects (e.g., tuples, attributes) are locked
  - Need more locks
  - More concurrency
- We can do both.
  - The bathroom metaphor:



Τέλος Ενότητας

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο
ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ & ΘΡΗΣΚΕΥΜΑΤΩΝ, ΠΟΛΙΤΙΣΜΟΥ & ΑΘΛΗΤΙΣΜΟΥ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ
ΕΣΠΑ 2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ
Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ακαδημαϊκά Μαθήματα στο Πανεπιστήμιο Κρήτης**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Ευρωπαϊκή Ένωση
Ευρωπαϊκό Κοινωνικό Ταμείο

ΕΠΙΧΕΙΡΗΣΙΑΚΟ ΠΡΟΓΡΑΜΜΑ
ΕΚΠΑΙΔΕΥΣΗ ΚΑΙ ΔΙΑ ΒΙΟΥ ΜΑΘΗΣΗ
επένδυση στην κοινωνία της γνώσης
ΥΠΟΥΡΓΕΙΟ ΠΑΙΔΕΙΑΣ ΚΑΙ ΘΡΗΣΚΕΥΜΑΤΩΝ
ΕΙΔΙΚΗ ΥΠΗΡΕΣΙΑ ΔΙΑΧΕΙΡΙΣΗΣ

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Σημειώματα

Σημείωμα αδειοδότησης

.

Σημείωμα Αναφοράς

Copyright Πανεπιστήμιο Κρήτης, Δημήτρης Πλεξουσάκης. «**Συστήματα Διαχείρισης Βάσεων Δεδομένων. Διάλεξη 9η: Transactions - part 2**». Έκδοση: 1.0. Ηράκλειο/Ρέθυμνο 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://www.csd.uoc.gr/~hy460/