ΕΛΛΗΝΙΚΗ ΔΗΜΟΚΡΑΤΙΑ
ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ

Συστήματα Διαχείρισης Βάσεων Δεδομένων

Διάλεξη 8η: Transactions - part 1

Δημήτρης Πλεξουσάκης
Τμήμα Επιστήμης Υπολογιστών

Transaction Management

- Transactions are units of work that must be executed atomically and (seemingly) in isolation from other transactions
- Their effects should be durable: no completed work should be lost
- ACID properties of transactions
  - atomicity, consistency, isolation, durability
- The transaction processor accepts transaction commands from applications and performs the tasks of:
  - logging for resilience / recovery
  - concurrency control
  - deadlock resolution
- This lecture: coping with failures

Database Consistency

- Would like data to be "accurate" or "correct" at all times
- Integrity constraints: predicates that data must satisfy
  - Types of constraints:
    - Keys
    - Functional dependencies / multi-valued dependencies
    - Domain constraints
    - Arbitrary predicates, e.g., no employee should earn more than twice the average salary
  - Constraints can be static or dynamic
- Consistency: a database is in a consistent state when all integrity constraints are satisfied
- A transaction is a collection of actions that preserve consistency, i.e. map a consistent DB state to another consistent DB state

Constraint Violation

- Constraint violations may occur due to:
  - transaction bugs
  - DBMS bugs
  - hardware failure
    - e.g., disk failure alters an attribute value
  - data sharing
    - e.g., transaction T1 gives a 10% raise to all employees of category A ; T2 promotes employees of category A to category B
- In many cases, constraint violations can be prevented
- In other cases, constraint violations need to be fixed in order to restore consistency

Recovery

- Many issues involved:
  - how to write correct transactions
  - how to check constraints
  - how to correct constraints
- We will deal with the problem of recovery from failures only
- Must first establish a failure model
- Failures are caused by undesired (expected or unexpected) events
  - Expected undesired events: memory loss, CPU halt, reset, …
  - Unexpected undesired events: everything else (e.g., disk damage)

Operations

- Input (x): block with x → memory
- Output (x): block with x → disk
- Read (x,t): do input(x) if necessary; t ← value of x in block
- Write (x,t): do input(x) if necessary; value of x in block ← t
- Unfinished transactions create consistency problems. E.g.,

  Constraint: A=B; Initially: A=B=1

  Transaction T1: A ← A × 2; B ← B × 2

  T1 is implemented as the sequence: Read (A,t); t ← t×2; Write (A,t); Read (B,t); t ← t×2; Write (B,t); Output (A); Output (B);

  Suppose failure occurs after Output (A);
  In memory: A=B=2
  On disk: A=2, B=1

Logging

- Previous example demonstrates the need for atomicity: either a transaction executes in its entirety or not at all
- One solution to the previous problem can be provided by logging
- A log is a sequence of log records, each recording something about the operations performed by a transaction
- The log is used to reconstruct what transactions were doing when a failure occurred
  - some transactions will be redone
  - others will be undone and the database will be restored (as if they never executed)
- Undo logging refers to the second type of repair

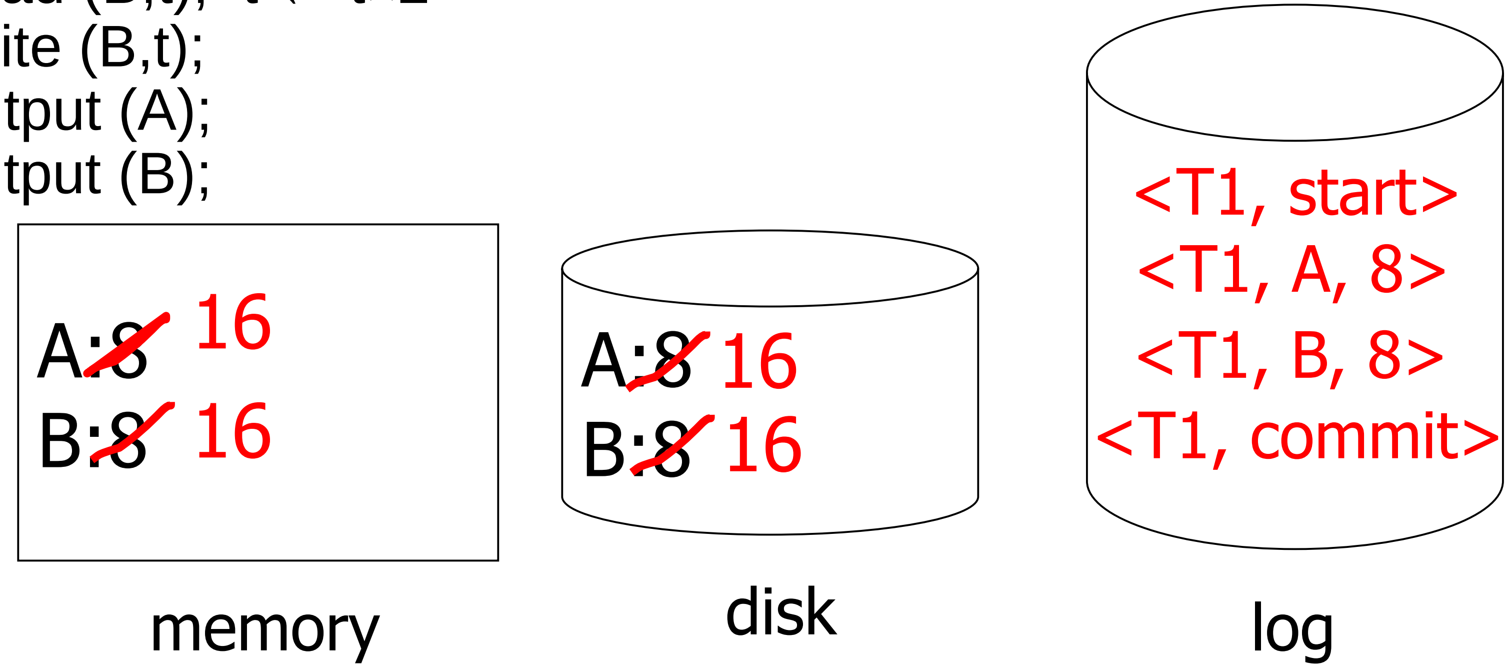
Undo Logging

- Logs are organized as append-only files
- Log blocks are initially created in main memory and are allocated by the buffer management like any other blocks
- Log blocks are written in 2-ary storage when feasible
- Forms of log records:
  - <START T> : transaction T has begun
  - <COMMIT T>: T has completed successfully; changes made by T should appear on disk
  - <ABORT T>: T could not complete successfully; changes made by T should not appear on disk
  - update records: <T, X, v>: T has changed X whose former value was v (such a record is written in response to a Write action, not an Output action)

Undo Logging

- Example: constraint A=B, initially A=B=8

```
T1:     Read (A,t);  t ← t×2
        Write (A,t);
        Read (B,t);  t ← t×2
        Write (B,t);
        Output (A);
        Output (B);
```



A:~~8~~ 16
B:~~8~~ 16

memory

A:~~8~~ 16
B:~~8~~ 16

disk

<T1, start>
<T1, A, 8>
<T1, B, 8>
<T1, commit>

log

Undo-Logging Rules

- Transactions must obey the following rules in order for undo logging to enable recovery from failure
  1. If T modifies element X, then the log record <T,X,v> must be written to the log before the new value of X is written to disk
  2. If T commits, then <COMMIT T> is written to disk only after all elements altered by T have been written to disk, but as soon after that as possible
- These rules impose the following order for writing on disk:
  1. Log records indicating the changed DB elements
  2. Changed DB elements themselves
  3. COMMIT log record
- To force log records to disk, the log manager needs a flush log command instructing the buffer manager to copy on disk log blocks that have not been copied or were recently changed

Undo-Logging Complications

- Logging actions do not occur in isolation; since many transactions execute simultaneously, log records for one transaction may be interleaved with similar operations for other transactions
- Flushing the log may imply that log records for a transaction appear on disk earlier than intended
- No harm as long as the <COMMIT T> record is written only after the output actions of T are completed
- If elements A and B share a block, writing one of them, implies writing the other as well (rule 1 may be violated)
- Need to impose additional constraints or impose a concurrency protocol
  - more on concurrency control later

Recovery using Undo-Logging

- Recovery manager must use the log to restore the database state into a consistent one in the event of a failure
- A simple recovery manager:
  - examine the entire log
  - make the necessary changes to the database
- The recovery manager needs to divide the transactions into committed and uncommitted ones
- If a record <COMMIT T> is found, then because of rule 2, all changes made by T must have already been written on disk. Hence, T has left the DB in a consistent state
- If a <START T> record is found but no <COMMIT T>, T is considered incomplete and must be undone

Recovery using Undo-Logging

- Undoing a transaction means reverting to the old values of the elements that were changed by the transaction
- Rule 1 ensures that if X was changed by T prior to the failure, there will be a record <T,X,v> in the log and this record must have been copied on disk before the failure
    - If T modifies element X, then the log record <T,X,v> must be written to the log before the new value of X is written to disk
- For recovery, the value v must be written for X
- Several transactions affecting the same element may have been left uncommitted. Systematic restoration of the values must take place (order must be respected)
- The log needs to be scanned backwards: from the most recent record to the earliest one.
- Must keep a list of transactions for which a <COMMIT > or an <ABORT > record is found.

Recovery using Undo-Logging

- If a record <T,X,v> is found:
  - if <COMMIT T> has been found, do nothing. No changes need to be undone
  - if T is incomplete, the value of X must be changed to v
- After making the necessary changes, a record <ABORT T> must be written for every incomplete transaction T and the log must be flushed
- This approach is simple but not very efficient:
  - in principle, the entire execution history must be examined every time a failure occurs
- An improvement of this relies on checkpointing the log in order to limit the extent to which the log must be examined

Checkpointing

- In order to reduce the size of the log to be examined every time a failure occurs, one could truncate the log after a transaction commits
  - however, if recovery becomes necessary , log records of other transactions could be lost as well
- Periodic checkpointing:
  1. Stop accepting new transactions
  2. Wait till all active transactions commit or abort and have written a <COMMIT > or <ABORT > record on the log
  3. Flush the log to disk
  4. Write a log record <CKPT> and flush the log
  5. Resume accepting transactions

Checkpointing

- No need to undo changes of transactions that executed prior to the checkpoint
  - (rule 2: If T commits, then <COMMIT T> is written to disk only after all elements altered by T have been written to disk, but as soon after that as possible)
- During recovery, scan the log backwards to identify incomplete transactions.
  - When a <CKPT> record is found, all incomplete transactions have been found
- No need to scan prior to the <CKPT> since no transaction may begin until checkpointing ends
- The log before the <CKPT> may actually be deleted or overwritten

Checkpointing

- Example: assume the log contains the following records

  <START T1>

  <T1, A, 5>

  <START T2>

  <T2, B, 10>

If we decide to place a checkpoint here, we must wait until T1 and T2 complete, before writing the <CKPT> record on the log file.

…..

<COMMIT T1>

<COMMIT T2>

<CKPT>

<START T3>

Nonquiescent Checkpointing

- Permits new transactions entering the system during checkpointing

1. Write log record <START CKPT(T1,T2, …Tk)> for the active transactions T1, T2, …, Tk and flush log
2. Wait until all of T1, T2, …, Tk commit or abort but do not prevent other transactions from starting
3. When all have finished, write log record <END CKPT> and flush log

Recovery with Nonquiescent Checkpointing

- Scan log file backwards:
  1. If <END CKPT> is encountered first, then all incomplete transactions must have started after the previous record <START CKPT (T1, … , Tk)>; scan backwards till the START CKPT record; previous log may be discarded
  2. If <START CKPT (T1, … , Tk)> is encountered first, crash has occurred during the checkpoint; incomplete transactions are those that we find while scanning backwards before we find the <START CKPT (T1, … , Tk)> record and those among T1, T2, .. Tk that did not complete.

     - no need to scan further back from the start of the earliest of these transactions

Recovery with Nonquiescent Checkpointing

- Example:assume the log contains the following records

<START T1>

<T1, A, 5>

<START T2>

<T2, B, 10>

For doing nonquiescent checkpointing, we need to write a log record

<START CKPT(T1,T2)>

T3 begins while waiting for T1 and T2 to complete

<T2, C, 15>

<START T3>

<T1, D, 20>

<COMMIT T1>

<T3, E, 25>

<COMMIT T2>

<END CKPT>

<T3, F, 30>

Assume that failure occurs at this point.

Recovery with Nonquiescent Checkpointing

- Example (cont'd)
  - T3 is the only incomplete transaction and F's value must be restored to 30
  - When we encounter the <END CKPT> we know that all incomplete transactions started after the previous <START CKPT> record
  - Scanning backwards we find that E's value must be restored to 25
  - There are no other transactions that started but did not commit
  - No scanning needed earlier than the <START CKPT> record

Recovery with Nonquiescent Checkpointing

- Example (cont’d): failure occurs during checkpoint

<START T1>
<T1, A, 5>
<START T2>
<T2, B, 10>
<START CKPT(T1,T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<COMMIT T1>
<T3, E, 25>

T3 and T2 are now incomplete and their changes must be undone.

When we find <START CKPT(T1,T2)>, the only possibly incomplete one is T1, but <COMMIT T1> has been found.

Only need to go back till the <START T2> record restoring B to 10

Redo Logging

- Undo logging imposes the requirement for immediate backup of database elements to disk
- This requirement is lifted in redo logging:
  - redo logging ignores incomplete transactions and repeats the changes made by committed transactions
  - undo logging cancels the effects of incomplete transactions and ignores committed ones
  - redo logging requires that the <COMMIT > record appears on disk before any changed values appear on disk
  - recall: undo logging requires that the <COMMIT > record appears on disk only after changed values appear on disk
  - redo log records have a different meaning than undo log records: the new rather than the old values are needed in redo logging

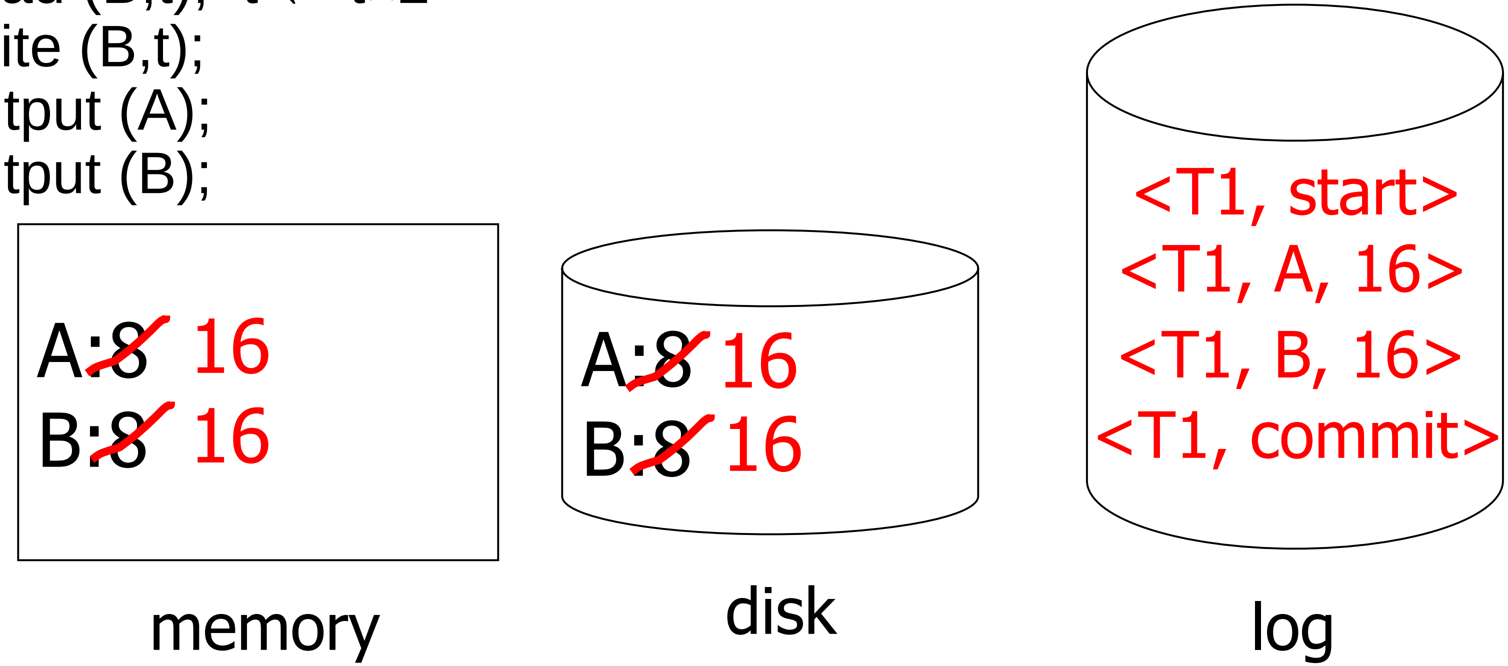
Redo-Logging Rules

- <T,X,v> : “transaction T wrote new value v for element X”
- Every time a transaction modifies a DB element X, a record of the form <T,X,v> must be written to the log
- The following rule (write-ahead log rule) determines the order in which data and log records appear on disk:
  - Before modifying any DB element X on disk, it is necessary that all records pertaining to the modification of X (<T,X,v>, <COMMIT T>) must appear on disk
- The rule implies the following order for writing records on disk:
  1. Log records indicating changed elements
  2. COMMIT log record
  3. Changed elements themselves

Redo Logging

- Example: constraint A=B, initially A=B=8

```
T1:     Read (A,t);  t ← t×2
        Write (A,t);
        Read (B,t);  t ← t×2
        Write (B,t);
        Output (A);
        Output (B);
```



Recovery using Redo-Logging

- Incomplete transactions can be treated as if they never occurred
  - no changes to DB elements by transaction T have been written to disk unless the log contains a <COMMIT T> record
- What about committed transactions?
  - we do not know which of their changes have appeared on disk
  - but we do not really care: the new values must be written on disk even if they are already there
- When failure occurs do:
  1. Identify committed transactions
  2. Scan log forward from the beginning and for each <T,X,v>:
     a. If T is not committed, do nothing
     b. If T is committed, write v for X
  3. For each incomplete transaction T, write <ABORT T> to log and flush log

Recovery using Redo-Logging

- Example:

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

If failure occurs after this point, <COMMIT> is on disk, hence T is complete. Values for A and B will be written on disk (even if it is not necessary)

Recovery using Redo-Logging

- Example:

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

If failure occurs between these points, <COMMIT> is not on disk, hence T is incomplete. No values for A and B will be written on disk and <ABORT T> will be written to the log

Recovery using Redo-Logging

- Example:

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

If failure occurs between these points, <COMMIT> may not be on disk.

If it is, T is complete. Otherwise, T is incomplete.

Checkpointing with Redo Logging

- With redo logging changes incurred by a completed transaction may appear on disk much later than the transaction's commit point
- If checkpointing is implemented with redo logging, all transactions and not just the active ones must be considered
- All database elements that have been modified by committed transactions but not written on disk, must be written during checkpointing
- Need to know which buffers are dirty, i.e., changed but not copied on disk
- Also need to know the transaction that modified each buffer

Nonquiescent Checkpointing with Redo Logging

- However, the checkpoint can be completed without waiting for transactions to complete or abort since they cannot have their changes written on disk
- Nonquiescent checkpointing proceeds as follows:

1. Write log record <START CKPT(T1,T2,…,Tk)> for the uncommitted transactions T1, T2, …, Tk and flush the log
2. Write to disk all elements written to buffers (but not yet to disk) by transactions already committed when the record <START CKPT(T1,T2,…,Tk)> was written to the log
3. Write <END CKPT> to the log and flush

Nonquiescent Checkpointing with Redo Logging

- Example:

<START T1>

<T1, A, 5>

<START T2>

<COMMIT T1>

<T2, B, 10>

<START CKPT(T2)>

<T2, C, 15>

<START T3>

<T3, D, 20>

<END CKPT>

<COMMIT T2>

<COMMIT T3>

Value of A may be on disk;

If not, A must be copied to disk before the checkpoint can end

Recovery with Checkpointed Redo Log

- When scanning the log file, the search can be limited by the start and end of the checkpoint

1. If the last CKPT record in the log before failure is <END CKPT>, then every value written by a transaction committed before the corresponding <START CKPT(T1,T2,…, Tk)> is on disk;

   Any transaction among T1,T2,…,Tk or any that started after the <START CKPT> may have changes that do not appear on disk and recovery proceeds as before

   No need to look further back than the earliest of the <START Ti> records

Recovery with Checkpointed Redo Log

2. If the last CKPT record in the log before failure is <START CKPT(T1,T2,…, TK)> , then it is not certain that transactions that committed before the start of the checkpoint have had their changes written to disk

   Must search backwards to the previous <END CKPT>, find its matching <START CKPT(T1,T2,…, Tk)> and redo changes as in case 1.

Recovery with Checkpointed Redo Log

- Example:

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

If failure occurs at the end, T2 and T3 must be redone; search log backwards up to <START T2> and write values 10,15,20 for B,C,D resp.

If failure occurs between <COMMIT T2> and <COMMIT T3>, then no change must be made to D and <ABORT T3> is added to the log

If failure occurs before <END CKPT> we must search back to the previous <END CKPT> and its corresponding <START CKPT> record; no such record exists here and T1 is the only committed transaction that is redone; <ABORT T2> and <ABORT T3> are written to the log

Study

- Garcia-Mollina, Ullman, Widom, “Database System Implementation”, chapter 8
- Ramakrishnan, Gehrke, “Συστήματα Διαχείρησης Βάσεων Δεδομένων”, κεφ. 20

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

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

ΕΣΠΑ
2007-2013
πρόγραμμα για την ανάπτυξη
ΕΥΡΩΠΑΪΚΟ ΚΟΙΝΩΝΙΚΟ ΤΑΜΕΙΟ

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 8η: Transactions - part 1**». Έκδοση: 1.0. Ηράκλειο/Ρέθυμνο 2015. Διαθέσιμο από τη δικτυακή διεύθυνση: http://www.csd.uoc.gr/~hy460/