COURSE PART C: Transaction Processing

Lecture 13

Transaction Management Overview

Chapter 17.1-17.3 and 17.6: Elmasri & Navathe, 5ED
Chapter 16.1-16.3 and 16.6: Ramakrishnan & Gehrke, 3ED

Demetris Zeinalipour

http://www.cs.ucy.ac.cy/~dzeina/courses/epl446
Overview of Transaction Processing (Επισκόπηση Επεξεργασίας Δοσοληψιών)

- We will now focus on the 3rd part of this course that deals with **Concurrency Control** (Έλεγχο Ταυτοχρονίας), and **Recovery Management** (Τεχνικές Ανάκαμψης) in cases of failures.
Lecture Outline
Transaction Management Overview

• 16.0) Introduction to Transactions (Δοσοληψίες ή Συναλλαγές)
• 16.1) The ACID (Atomicity-Consistency-Isolation-Durability) Properties
• 16.2) Transactions and Schedules (Χρονοπρόγραμμα)
• 16.3) Concurrent Executions of Transactions (Ταυτόχρονες Εκτελέσεις Δοσοληψιών) and Problems
• 16.6) Transaction Support in SQL

Below topics will be covered as part of subsequent lectures
• 16.4) Concurrency with Locks (Κλειδαρίες)
• 16.5) Concurrency with Timestamps (Χρονόσημα)
• 16.7) Introduction to Crash Recovery (Ανάκαμψη Σφαλμάτων)
Introduction to Transactions
(Εισαγωγή σε Δοσοληψίες)

• The concept of transaction (δοσοληψία) provides a mechanism for describing logical units of database processing.
  – Analogy: A transaction is to a DBMS as a process is to an Operating System.

• Transaction Processing Systems (Συστήματα Επεξεργασίας Δοσοληψιών) are systems with large databases and hundreds of concurrent users executing database transactions
  – Examples: Airline Reservations (Αεροπορικές Κρατήσεις), Banking (Εφαρμογές Τραπεζικού Τομέα), Stock Markets (Χρηματιστήρια), Supermarkets (Υπεραγορές),
Introduction to Transactions
(Εισαγωγή σε Δοσοληψίες)

- **Transaction (Δοσοληψία) (Xact)**, is an **atomic** (i.e., all-or-nothing) sequence of **database operations** (i.e., read-write operations).

- It is the **DBMS’s abstract view of a user program**!

- A **transaction** (collection of actions) makes **transformations** of system states while preserving the **database consistency** (συνέπεια βάσης) … next slide.

![Diagram of transaction execution]

- Begin transaction T
- DB in a consistent state
- Execution of a transaction T
- DB may be in an inconsistent state during execution
- End transaction T
- DB in a consistent state
Introduction to Transactions (Εισαγωγή σε Δοσοληψίες)

- One way of specifying the transaction boundaries is by specifying explicit BEGIN TRANSACTION and END TRANSACTION statements in an application program.

  - **Transaction Example in MySQL**
    
    ```
    START TRANSACTION;
    SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
    UPDATE table2 SET summary=@A WHERE type=1;
    COMMIT;
    ```

  - **Transaction Example in Oracle (same with SQL Server)**
    
    - When you connect to the database with `sqlplus` (Oracle command-line utility that runs SQL and PL/SQL commands interactively or from a script) a transaction begins.
    - Once the transaction begins, every SQL DML (Data Manipulation Language) statement you issue subsequently becomes a part of this transaction.

  - **Transaction Example in C: See Next Slide**

    Note that the given example has no explicit START/END statements as the whole program is essentially 1 transaction (as the previous example with Oracle’s sqlplus utility).
Transaction Consistency
Example with Embedded SQL

- The below example shows a Transaction constraint (not captured by ICs)

```c
main {
...
EXEC SQL BEGIN DECLARE SECTION; // define C host program variables (accessible in SQL environment)
char flight_no[6], customer_id[20]; // these host-language variable are prefixed with ":" in SQL statements
char day;
EXEC SQL END DECLARE SECTION;
scanf("%s %d %s", flight_no, day, customer_id);
EXEC SQL UPDATE FLIGHT
SET STSOLD = STSOLD + 1
WHERE FNO = :flight_no AND DATE = :day;
EXEC SQL INSERT INTO FC(FNO, DATE, CID, SPECIAL);
VALUES(:flight_no, :day, :customer_id, null);
printf("Reservation completed");
return(0);
}
```

Consider an airline reservation example with the relations*:
**FLIGHT(FNO, DATE, SRC, DEST, STSOLD)**
**CUST(CID, ADDR)**
**FC(FNO, DATE, CID, SPECIAL)**

* We make some **simplifying assumptions** regarding the schema and constraints

---

**Sell a seat** on a given flight and date by increasing the **SeaTSOLD** attribute

**Store the sale** in the Flight-Customer table

If only the first action is executed then relations **FLIGHT and FC** will be inconsistent

⇒ Although **not a concurrent program**, we need to ensure **transaction consistency** (all-or-nothing)!

---

* We make some **simplifying assumptions** regarding the schema and constraints
**DB Consistency vs. Trans. Consistency**

(Συνέπεια ΒΔ vs. Συνέπεια Δοσοληψίας)

- **Database Consistency (Συνέπεια Βάσης)**
  - A database is in a consistent state if it **obeys all of the Integrity Constraints** (Κανόνες Ακεραιότητας) defined over it.
  - Examples of Integrity Constraints:
    - **Domain Constraints** (Πεδίου Ορισμού): e.g., SID must be integer
    - **Key Constraints** (Κλειδιού): e.g., no 2 students have the same SID
    - **Foreign Key Constraints** (Ξένου Κλειδιού): e.g., DepartmentID in Student must match the DepartmentID in Department’s table.
    - **Single Table Constraints**: e.g., CHECK (age>18 AND age<25)
    - **Multiple Table Constraints**:
      - e.g., Create ASSERTION C CHECK ((SELECT A) + (SELECT B) < 10)

- **Transaction consistency (Συνέπεια Δοσοληψίας)**
  - Complements Database consistency by incorporating user semantics (e.g., σημασιολογία όπως ορίζεται από τον χρήστη)
  - T.C is the user’s responsibility (e.g., previous example)
Introduction to Transactions
(Εισαγωγή σε Δοσοληψίες)

State Diagram for Transaction Execution
(Διάγραμμα Καταστάσεων για την Εκτέλεση Δοσοληψιών)

**Active:** When Xact begins

**Partially Committed:** When Xact ends, several recovery checks take place making sure that the DB can always recover to a consistent state

**Failed:** If Xact aborts for any reason (rollback might be necessary to return the system to a consistent state)

**Committed:** After partial committed checks are successful. Once committed we never return (roll-back) to a previous state
Introduction to Transactions
(Εισαγωγή σε Δοσοληψίες)

• Things get even more complicated if we have several DBMS programs (transactions) executed concurrently.

• Why do we need concurrent executions?
  – **It is essential** for good DBMS performance!
    • Disk accesses are frequent, and relatively slow
    • Overlapping I/O with CPU activity increases throughput and response time.

• What is the problem with concurrent transactions?
  – **Interleaving (Παρεμβάλλοντας)** transactions might lead the system to an inconsistent state (like previous example):
    • **Scenario:** A Xact prints the monthly bank account statement for a user U (one bank transaction at-a-time). **Before** finalizing the report another Xact **withdraws $X** from user U.
    • **Result:** Although the report contains an updated final balance, it shows nowhere the bank transaction that caused the decrease (unrepeatable read problem, explained next)

• A DBMS guarantees that these problems will not arise.
  – **Users are given the impression that the transactions are executed sequentially (σειριακά), the one after the other.**
The ACID properties (Οι ιδιότητες ACID)

- What are the fundamental (θεμελιώδεις) properties that a DBMS must enforce so that data remains consistent (in the face of concurrent access & failures)?
- A DBMS needs to enforce four (4) properties:
  Atomicity – Consistency – Isolation - Durability
  Ατομικότητα – Συνέπεια - Απομόνωση – Μονιμότητα

- Jim Gray defined the key transaction properties of a reliable system in the late 1970.
- Acronym ACID was coined by Reuter and Haerder in 1983
The ACID properties
(Οι ιδιότητες ACID)

1. **Atomicity (Ατομικότητα):** All or nothing!
   - An executing transaction completes in its **entirety** (i.e., ALL) or it is aborted altogether (i.e., NOTHING).
   - e.g., Transfer_Money(Amount, X, Y) means i) DEBIT(Amount, X); ii) CREDIT(Amount, Y). Either both take place or none.
   - Reasons for Incomplete Transactions
     - Anomaly Detection (e.g., Constraint violation) or System Crash (e.g., power)
   - Responsibility: Recovery Manager (use log file to record all writes)

2. **Consistency (Συνέπεια):** Start & End Consistent!
   - If each Transaction is consistent, and the DB starts consistent, then the Database ends up consistent.
   - If a transaction violates the database’s consistency rules, the entire transaction will be rolled back and the database will be restored to a state consistent with those rules
   - Responsibility: User (DB only enforcing IC rules)
The ACID properties
(Oι ιδιότητες ACID)

3. Isolation (Απομόνωση): See your own data only!
   - An executing transaction cannot reveal its (incomplete) results before it commits.
   - Consequently, the net effect is identical to executing all transactions, the one after the other in some serial order.
     • e.g., if two transactions T1 and T2 exists, then the output is guaranteed to be either T1, T2 or T2, T1 (The DBMS cannot guarantee the order of execution, that is the user’s job!) … see example next page
   - Responsibility: Lock Manager (i.e., Concurrency Control Manager)

4. Durability (Μονιμότητα): DBMS Cannot Regret!
   - Once a transaction commits, the system must guarantee that the results of its operations will never be lost, in spite of subsequent failures.
   - Responsibility: Recovery Manager (use log file to record all writes)
Notation for Transactions
(Σημειογραφία για Δοσοληψίες)

- **Actions** executed by a transaction include **reads** and **writes** of **database objects**

**Notation**
- \( R_T(O) \): The Transaction \( T \) **Reads** an Object \( O \).
- \( W_T(O) \): The Transaction \( T \) **Writes** an Object \( O \).
  - When Transaction is clear in context we shall omit the \( T \)
  - Although written, the data is in really **pending** until **committed**.
- **Commit\(_T\)**: Complete successfully **writing** data to disk
- **Abort\(_T\)**: Terminate and **undo** all carried out actions

**Assumptions**
- Transaction **Communication** only **through the DBMS**
- Database Objects: **Static Collection** (i.e., tables, etc. not added/removed ... dynamic case more complex)
Transactions and Schedules
(Δοσοληψίες και Χρονοπρόγραμμα)

- **Schedule (Χρονοπρόγραμμα)**
  - List of *actions* (read, write, abort, or commit) from a *set* (ομάδας) of transactions (T1, T2, …) where the *order of actions* inside each transaction does not change.
    - e.g., if T1=R(A), W(A) then W(A), R(A) is not a schedule (as it is in opposite order)

<table>
<thead>
<tr>
<th>Schedule</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
</tr>
<tr>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
</tr>
<tr>
<td>R(C)</td>
</tr>
<tr>
<td>W(C)</td>
</tr>
</tbody>
</table>

- Note that the DBMS might carry out *other actions* as well (e.g., evaluation arithmetic expressions).
- Yet these do not affect the other transactions, thus will be omitted from our presentation
- We shall introduce **Commits/Aborts** in the next lecture
Transactions and Schedules
(Δοσοληψίες και Χρονοπρόγραμμα)

• **Serial Schedule (Σειριακό Χρονοπρόγραμμα)**
  – A schedule in which the different transactions are NOT interleaved (i.e., transactions are executed from start to finish one-by-one)

<table>
<thead>
<tr>
<th>Serial Schedule</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(B)</td>
<td>W(A)</td>
<td></td>
</tr>
<tr>
<td>R(B)</td>
<td>W(B)</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Serial Schedule</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(B)</td>
<td>W(B)</td>
<td></td>
</tr>
<tr>
<td>R(A)</td>
<td>W(A)</td>
<td></td>
</tr>
</tbody>
</table>
Problems due to Interleaved Xact
(Προβλήματα από την Παρεμβολή Δοσοληψιών)

Problems that arise when interleaving Transactions.

• **Problem 1: Reading Uncommitted Data (WR Conflicts)**
  - Reading the value of an uncommitted object might yield an inconsistency
  - **Dirty Reads** or **Write-then-Read (WR) Conflicts**.
  - In Greek: Ασυνεπείς αναγνώσεις

• **Problem 2: Unrepeatable Reads (RW Conflicts)**
  - Reading the same object twice might yield an inconsistency
  - **Read-then-Write (RW) Conflicts** (ή Write-After-Read)
  - In Greek: Μη-επαναλήψιμες αναγνώσεις

• **Problem 3: Overwriting Uncommitted Data (WW Conflicts)**
  - Overwriting an uncommitted object might yield an inconsistency
  - **Lost Update** or **Write-After-Write (WW) Conflicts**.
  - In Greek: Απώλειες ενημερώσεων

• **Remark:** There is no notion of **RR-Conflict** as no object is changed.
Reading Uncommitted Data (WR Conflicts) (Ασυνεπείς αναγνώσεις)  
“Reading the value of an uncommitted object yields an inconsistency”

- To illustrate the **WR-conflict** consider the following problem:
  
  **T1:** Transfer $100 from Account A to Account B 
  **T2:** Add the annual interest of 6% to both A and B.

---

**Problem caused by the WR-Conflict?** Account B was credited with the interest on a smaller amount (i.e., 100$ less), thus the result is not equivalent to the serial schedule
Unrepeatable Reads (RW Conflicts) 
(Μη-επαναλήψιμες αναγνώσεις) 
“Reading the same object twice yields an inconsistency”

- To illustrate the **RW-conflict** consider the following problem: 
  **T1**: Print Value of A  
  **T2**: Decrease Global counter A by 1.

<table>
<thead>
<tr>
<th>Trace</th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(A)</td>
<td></td>
<td>Write-after-Read</td>
</tr>
<tr>
<td>R(A)</td>
<td></td>
<td>R(A)</td>
</tr>
<tr>
<td>W(A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Problem caused by the RW-Conflict?**
Although the “A” counter is read twice within T1 (without any intermediate change) it has two different values (unrepeatable read)! … what happens if T2 aborts? 

**T1** has shown an incorrect result.

Note that if I read at this point we would see 9 rather than 10 (i.e., read is unrepeatable)
Overwriting Uncommitted Data (WW Conflicts)  
(Μη-επαναλήψιμες αναγνώσεις)  
“Overwriting an uncommitted object yields an inconsistency”

- To illustrate the WW-conflict consider the following problem:

   **Constr: Salary** of employees A and B must be kept equal

   **T1:** Set Salary to 1000; **T2:** Set Salary equal to 2000

---

**Overwriting an uncommitted object yields an inconsistency**

---

* Note: Client certainly prefers the 1st than the 2nd schedule (final amount is larger)!

**Problem caused by the WW-Conflict?**

Employee “A” gets a salary of 2000 while employee “B” gets a salary of 1000, thus result is not equivalent to the serial schedule!
WR Conflict (dirty read): A transaction $T_2$ could read a database object $A$ that has been modified by another transaction $T_1$, which has not yet committed.

RW Conflict (unrepeatable read): A transaction $T_2$ could change the value of an object $A$ that has been read by a transaction $T_1$, while $T_1$ is still in progress.

WW Conflict (lost update): A transaction $T_2$ could overwrite the value of an object $A$, which has already been modified by a transaction $T_1$, while $T_1$ is still in progress.
More Examples on Conflicts (for home) (Περισσότερα Παραδείγματα Συγκρούσεων)

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_item($X$); $X := X - N$;</td>
<td>read_item($X$); $X := X + M$;</td>
</tr>
<tr>
<td>write_item($X$);</td>
<td>Lost Update</td>
</tr>
<tr>
<td>read_item($Y$);</td>
<td></td>
</tr>
<tr>
<td>$Y := Y + N$;</td>
<td>write_item($X$);</td>
</tr>
<tr>
<td>write_item($Y$);</td>
<td></td>
</tr>
</tbody>
</table>

Unrepeatable Read
More Examples on Conflicts (for home) (Περισσότερα Παραδείγματα Συγκρούσεων)

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_2$</th>
</tr>
</thead>
</table>
| read_item($X$);  
$X := X - N$;  
write_item($X$), | Dirty Read            |
| read_item($Y$); |                        |

Lost Update:  
$X := X + M$;  
write_item($X$);
More Examples on Conflicts (for home) 
(Περισσότερα Παραδείγματα Συγκρούσεων)

<table>
<thead>
<tr>
<th>$T_1$</th>
<th>$T_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_item($X$); $X := X - N$; write_item($X$);</td>
<td>$sum := 0$; read_item($A$); $sum := sum + A$;</td>
</tr>
<tr>
<td>read_item($Y$); $Y := Y + N$; write_item($Y$);</td>
<td>:</td>
</tr>
</tbody>
</table>

Dirty Read

$read_item(X)$; $sum := sum + X$; $read_item(Y)$; $sum := sum + Y$;

Unrepeatable Read