Week 6

Transactions

Motivation

Motivated by 2 requirements of DBMS:

- Persistent – Outlive the programs that create/access the data
- Safe – hardware/software failures, malicious users
- Multi-user – concurrently access to data (concurrency control)
- Convenient
  - Physical Data Independence; huge difference between physical representation of data on disk and the logical way of seeing and working with;
  - High level, declarative (what, not how) query languages (e.g. SQL)
- Efficient – thousands of operations (query/update) per second
- Reliable – 99.99999 % uptime

Concurrent Access: Attribute level inconsistency

- Example: 2 users concurrently modify Students table:
  - User1:
    - `UPDATE Students SET Priority=Priority+1 WHERE ID=123456789`
  - User2:
    - `UPDATE Students SET Priority=Priority+0.5 WHERE ID=123456789`
- The computation `Priority = Priority+X` is decomposed into 3 basic operations by the system:
  - Step 1: the system fetches the value of field `Priority` from the database
  - Step 2: performs computation (increment the value with `X`)
  - Step 3: writes back the value of field `Priority` in the database
- Starting with `Priority=1`, what are the possible values
  - Sequential execution of the 2 statements
  - Inter-leaved execution of the 2 statements

Concurrent Access: Tuple level inconsistency

- Example: 2 users concurrently modify Students table:
  - User1:
    - `UPDATE Students SET Priority=1 WHERE ID=123456789`
  - User2:
    - `UPDATE Students SET Name='Popescu' WHERE ID=123456789`

- Similarly, each UPDATE statement is decomposed into 3 basic operations by the system:
  - Step 1: the system fetches the values of the whole tuple from the database
  - Step 2: performs computation (set a new value)
  - Step 3: writes back the whole tuple in the database

- What are the possible values
  - Sequential execution of the 2 statements
  - Inter-leaved execution of the 2 statements

Concurrent Access: Table level inconsistency

- Example: 2 users concurrently:
  - User1:
    - ```
      UPDATE Enrollments
      SET decision='Y'
      WHERE CNP IN (SELECT CNP FROM Students
                    WHERE TotalCredits>50)
      ```
  - User2:
    - ```
      UPDATE Students
      SET TotalCredits=TotalCredits+10
      WHERE MajorCode='INFO'
      ```

Concurrent Access: Multi-statement inconsistency

- Example: 2 users concurrently:
  - User1:
    - `INSERT INTO Archive`
      `SELECT * FROM Enrollments WHERE Decision = 'N';`
    - `DELETE FROM Enrollments WHERE Decision = 'N';`
  - User2:
    - `SELECT COUNT(*) FROM Enrollments;`
    - `SELECT COUNT(*) FROM Archive;`

Concurrency

- The goal of concurrency is to execute sequence of SQL statements so they appear to be running in isolation (avoid inconsistent and unexpected behavior).

- We **need** concurrency so that DBMS offers better performance.

Resilience to System Failures

- What happens if a system failure (hw/sw) happens
  - during a bulk load of data (from a large file, for example) into a database? => partially loaded
  - During executing sequence of commands altering multiple tables (see example in multi-statement consistency example)
  - Lots of updates (committing from cache memory to disks)

- The goal of resilience is to guarantee all-or-nothing execution!

- The solution for concurrency and resilience to failures is….

Resilience to System Failures

- What happens if a system failure (hw/sw) happens
  - during a bulk load of data (from a large file, for example) into a database? => partially loaded
  - During executing sequence of commands altering multiple tables (see example in multi-statement consistency example)
  - Lots of updates (committing from cache memory to disks)

- The goal of resilience is to guarantee all-or-nothing execution!

- The solution for concurrency and resilience to failures is….

Transactions

DEF: A transaction is a sequence of one or more SQL statements treated as a **unit**!

- They appear to run in isolation. (concurrency)
- If the system fails, transaction's changes are reflected either entirely or not at all. (resilience)
- **All or Nothing**

- When a transaction begins/ends?
  - Begin transaction, commit, rollback commands
    - on `commit` transaction ends and a new one begins
  - Transaction begins automatically on first SQL statement and current transaction ends when session terminates
  - `autocommit` turns each SQL statement into a transaction

- Transaction can have 2 outcomes: committed or aborted (rolled back).
  - Database remains in consistent state in either state.

Transaction properties

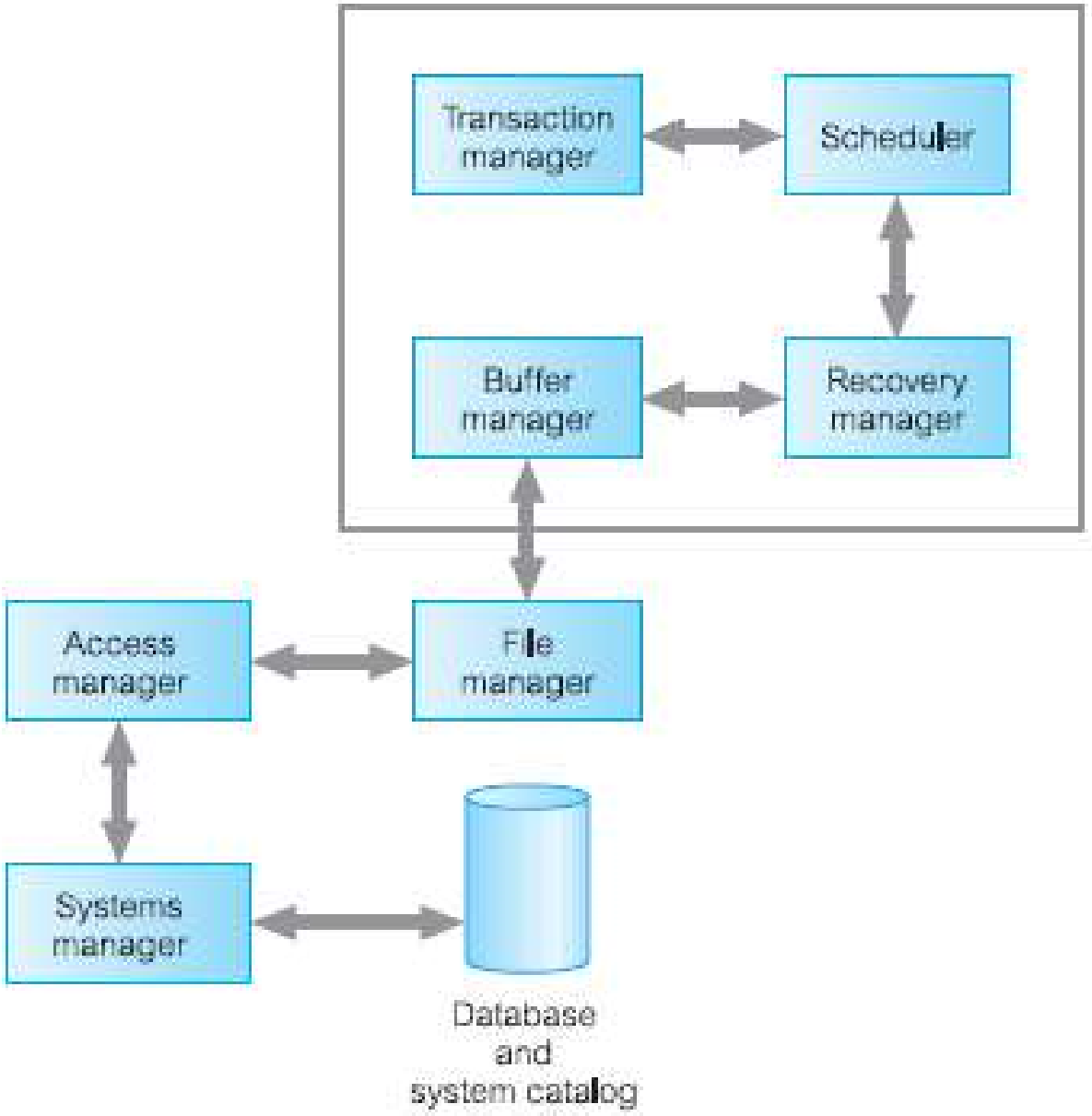
Atomicity,

Consistency,

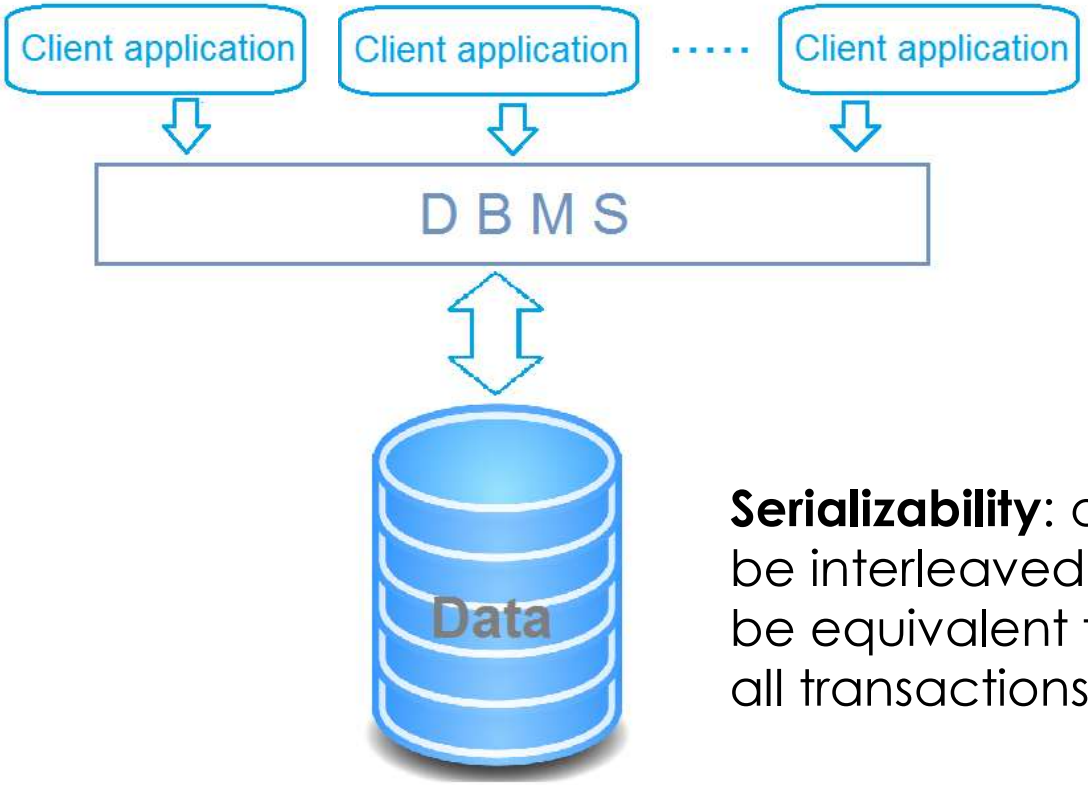
Isolation,

Durability

DBMS Transaction Subsystem



Isolation



Each client *i* issues a sequence of transactions $T_{i1},...T_{in}$, each transaction being composed of multiple SQL statements

Serializability: operations within transactions may be interleaved across clients, but execution must be equivalent to some sequential (serial) order of all transactions.

=> T_{11}, T_{12}, T_{21}, T_{13}, T_{22}, ...

How? By locking portions of the database

Concurrent Access: Attribute level inconsistency

- Example: 2 users concurrently modify Students table:
  - T1: `UPDATE Students SET Priority=Priority+1 WHERE ID=123456789`
  - T2: `UPDATE Students SET Priority=Priority+0.5 WHERE ID=123456789`
- => equivalent sequential executions: T1; T2 or T2; T1. In either case, if we start with Priority=1, it's correctly updated to 2.5 either way

Concurrent Access: Tuple level inconsistency

- Example: 2 users concurrently modify Students table:
  - T1: `UPDATE Students SET Priority=1 WHERE ID=123456789`
  - T2: `UPDATE Students SET Name='Popescu' WHERE ID=123456789`
- => equivalent sequential executions: T1; T2 or T2; T1. In either case, both fields will be correctly updated.

Concurrent Access: Table level inconsistency

- Example: 2 users concurrently:
  - T1: `UPDATE Enrollments SET decision='Y' WHERE CNP IN (SELECT CNP FROM Students WHERE TotalCredits>50)`
  - T2: `UPDATE Students SET TotalCredits=TotalCredits+10 WHERE MajorCode='INFO'`
- T1;T2 -> what result?
- T2;T1 -> what result?

Concurrent Access: Multi-statement inconsistency

- Example: 2 users concurrently:
  - T1:
    - `INSERT INTO Archive SELECT * FROM Enrollments WHERE Decision = 'N';`
    - `DELETE FROM Enrollments WHERE Decision = 'N';`
  - T2:
    - `SELECT COUNT(*) FROM Enrollments;`
    - `SELECT COUNT(*) FROM Archive;`
- T1;T2 -> what result?
- T2;T1 -> what result?

Durability

A client issues a sequence of transactions T_1,…T_n, each transaction being composed of multiple SQL statements.

Durability guarantees if system crashes after transaction commits, all effects of transaction remain in the database.

How? Using logging.

Atomicity

A client issues a sequence of transactions T_1,…T_n, each transaction being composed of multiple SQL statements.

Atomicity : each transaction is “all or nothing” , never left half-done. i.e. if there’s a crash during the execution of statements composing a transaction T then the effects of executed statements are undone.

How? Using logging.

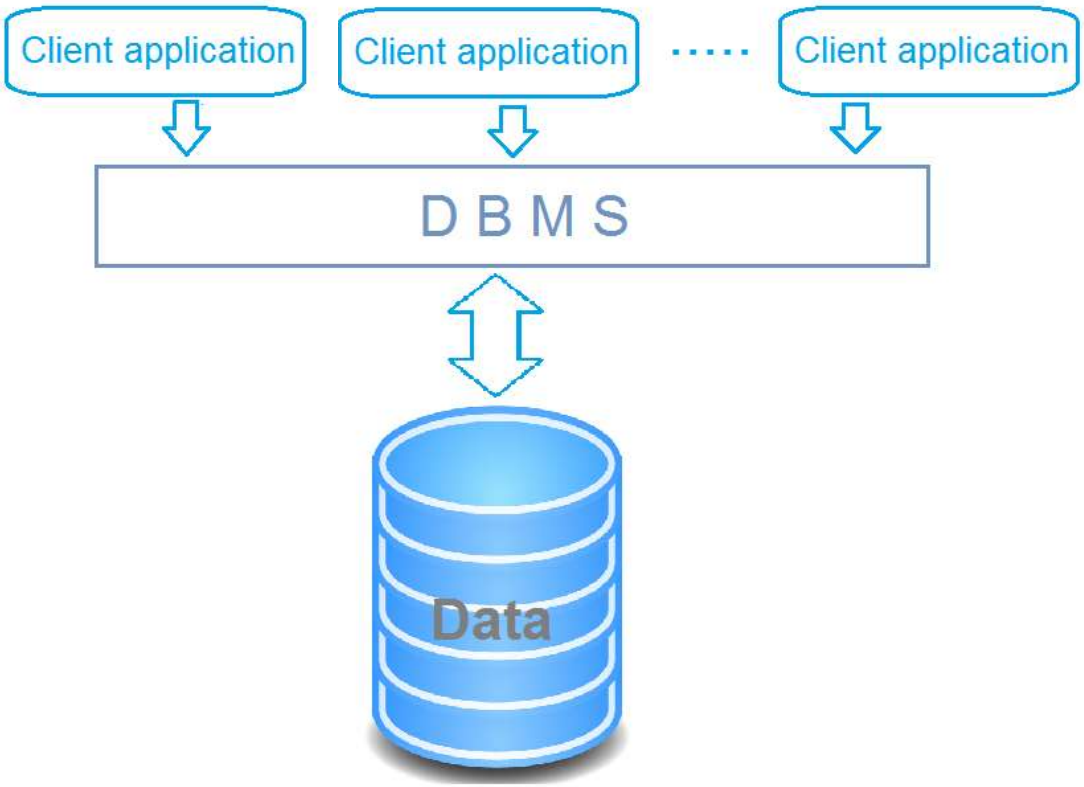
Remark: Application developers need to check the result returned by the DBMS for each executed transaction, and if there’s an error re-execute the transaction.

Transaction abort / rollback = undo of partial effects of a transaction

Transaction rollback can be initiated

- by system when there’s an error, or
- it can be issued by clients

Consistency



Each client *i* issues a serie of transactions $T_{i1},\ldots T_{in}$, each transaction being composed of multiple SQL statements

Consistency – how *integrity constraints* defined on a database interact with transactions.

Each client can assume that all constraints hold when a transaction begins, and
Each client must guarantee that all constraints hold when a transaction finishes.

Serializability => **constraints always holds**

$T_{11}, T_{12}, T_{21}, T_{13}, T_{22}, \ldots$

Exercise

Consider a relation R(A) containing two tuples {(2),(3)} and two transactions:

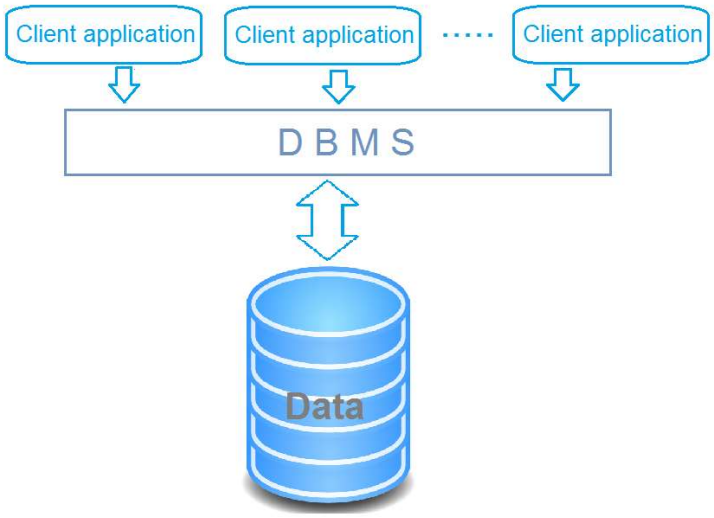
T1: Update R set A = A+1

T2: Update R set A = 2*A

Which of the following is NOT a possible final state of R?

a) 5, 6
b) 6, 8
c) 4, 6
d) 5, 7

Isolation levels



Each client *i* issues a serie of transactions $T_{i1},...T_{in}$, each transaction being composed of multiple SQL statements.

Serializability: operations within transactions may be interleaved across clients, but execution must be equivalent to some sequential (serial) order of all transactions.

Serializability drawbacks:

- Overhead (locking protocols)
- Reduced concurrency

This is why DBMS offer **Weaker Isolation Levels:**

- Read Uncommitted
- Read Committed
- Repeatable Read

Weaker

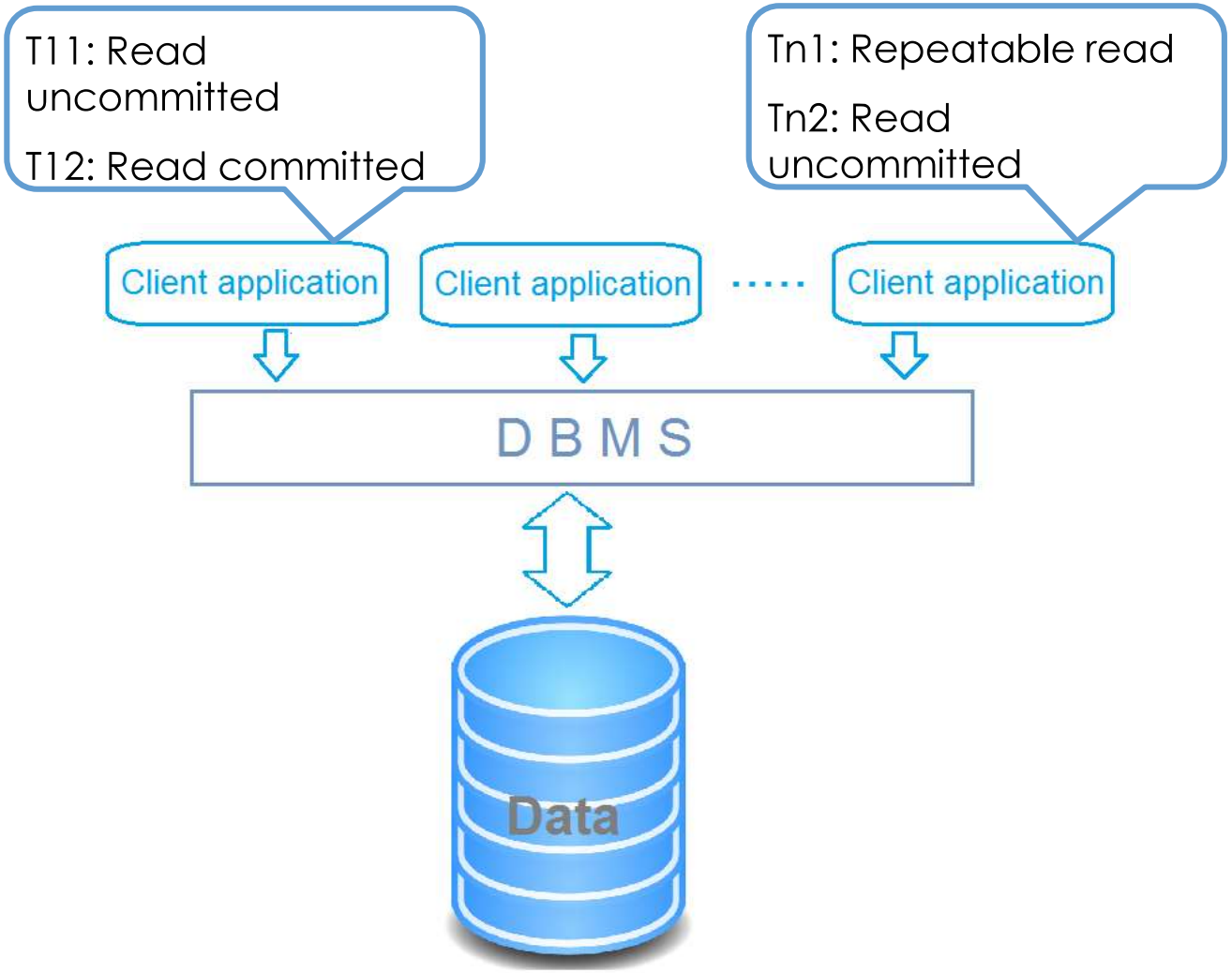
Stronger

⇐ Lower overhead, Increased concurrency, Lower consistency guarantees

- \+ Serializability (the default and strongest one)

Isolation levels

- Isolation level is per transaction
- Different transactions with different isolation levels may be executed concurrently



Dirty reads

DEF [‘Dirty’ data] A data item in the database is **‘dirty’** if it’s been written by a transaction that has not yet committed.

Example: two transactions executing concurrently

- T1: `UPDATE Students SET Priority=Priority+10 WHERE ID=123456789`

- T2: `SELECT AVG(Priority) FROM Students`

Read uncommitted

DEF: A transaction with **Read uncommitted** isolation level may perform dirty reads.

Example: two transactions executing concurrently

- T1: `UPDATE Students SET Priority=Priority+10 WHERE TotalCredits < 50`
- T2: `SELECT AVG(Priority) FROM Students`
  - With no isolation levels specified, the DBMS default isolation level is used
  - Serializable is in most cases the default, but there are exceptions, such as Oracle /MySQL that use Repeatable Read as default
  - With Serializability, the transactions will be executed either T1; T2 or T2; T1.

Read uncommitted

DEF: A transaction with **Read uncommitted** isolation level may perform dirty reads.

Example: two transactions executing concurrently

- T1: `UPDATE Students SET Priority=Priority+10 WHERE TotalCredits < 50;`
- T2: `SELECT AVG(Priority) FROM Students`

With no isolation levels specified, the Serializabilty is default, i.e. transactions will be executed either T1; T2 or T2; T1.

- T2:
```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
SELECT AVG(Priority) FROM Students;
```

We don’t have exact consistency => only an approximation of Priority average, not the exact value.

Exercise

Consider a table R(A) containing {(1),(2)}. Suppose transaction

T1: UPDATE R SET A = 2*A
T2: SELECT AVG(A) FROM R

If transaction T2 executes using "read uncommitted", what are the possible values it returns?

a) 1.5, 2, 3
b) 1.5, 2.5, 3
c) 1.5, 3
d) 1.5, 2, 2.5, 3

Read committed

DEF: A transaction with **Read committed** isolation level may **NOT** perform dirty reads.

Read committed

DEF: A transaction with **Read committed** isolation level may **NOT** perform dirty reads.

Example: two transactions executing concurrently

- T1:
```
UPDATE Students SET Priority=Priority+10 WHERE
TotalCredits < 50;
```

- T2:
```
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
    SELECT AVG(Priority) FROM Students;
    SELECT MAX(Priority) FROM Students;
```

Stronger, but it does not guarantees serializability.

We read only committed values, but two consecutive reads (e.g. AVG and MAX in T2) MAY return two different values, because another transaction (T1) might change the value. Thus, AVG may be computed before T1 and MAX is computed after T1.

Repeatable read

DEF: A transaction with **Repeatable read** isolation level may **NOT** perform dirty reads and if an item is read multiple times it cannot change value.

Repeatable read

DEF: A transaction with **Repeatable read** isolation level may **NOT** perform dirty reads and if an item is read multiple times it cannot change value.

Example: two transactions executing concurrently

- T1:
```
UPDATE Students SET Priority=Priority+10;
UPDATE Students SET TotalCredits=60 WHERE CNP=1..9;
```

- T2:
```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT AVG(Priority) FROM Students;
SELECT AVG(TotalCredits) FROM Students;
```

Stronger, but it does not guarantees serializability.

With Repeatable Read, when a value is read it is *locked* and can't be modified by other transactions.

Repeatable read. Phantom tuples

Repeatable read isolation level allows *a relation* to change value if read multiple times through **phantom tuples**.

- T1: `INSERT INTO Students [10 tuples];`

- T2:
```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT AVG(Priority) FROM Students;
SELECT MAX(Priority) FROM Students;
```

- T1: `DELETE FROM Students [10 tuples];`

Not allowed

Read only transaction

- Helps DBMS to optimize performance
- Independent of isolation level
- Example:

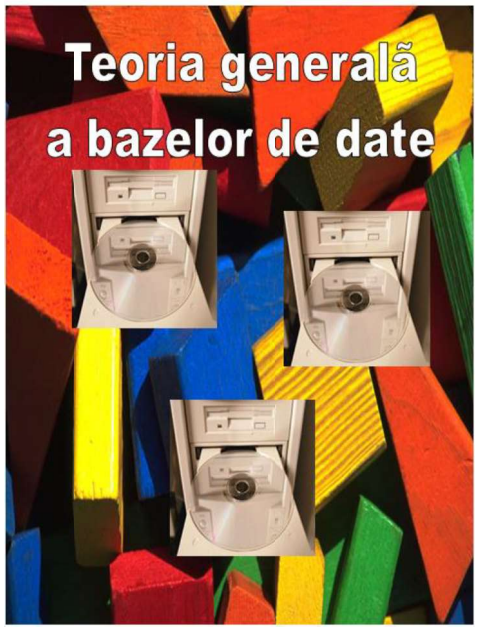
```
SET TRANSACTION READ ONLY;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT AVG(Priority) FROM Students;
SELECT MAX(Priority) FROM Students;
```

Isolation levels summary

	Dirty reads	Non-repeatable reads	Phantom tuples
Read uncommitted	Y	Y	Y
Read committed	N	Y	Y
Repeatable read	N	N	Y
Serializable	N	N	N

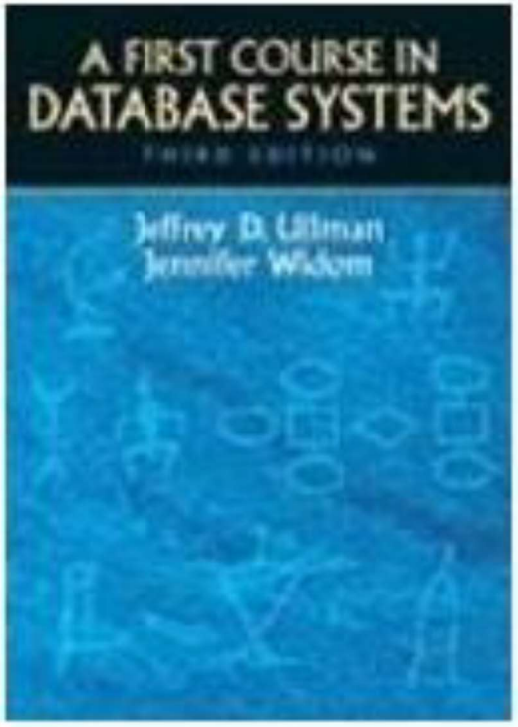
- Weaker isolation levels
  - Increased concurrency => better performance
  - Reduced overhead
  - Lower consistency

Bibliography (recommended)



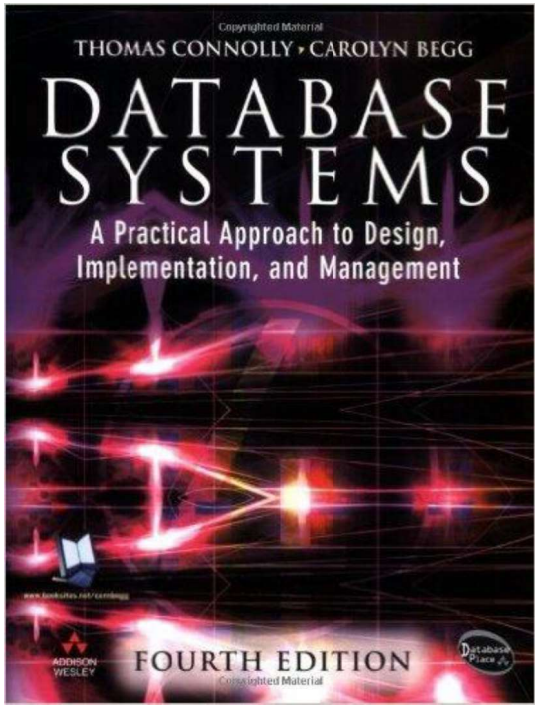
Teoria generala a bazelor de date, I. Despi, G. Petrov, R. Reisz, A. Stepan, Mirton, 2000

Cap 14.6 & 15.1



A First Course in Database Systems (3rd edition) by Jeffrey Ullman and Jennifer Widom, Prentice Hall, 2007

Chapter 6.5 - 6.6



Database Systems - A Practical Approach to Design, Implementation, and Management (4th edition) by Thomas Connolly and Carolyn Begg, Addison-Wesley, 2004

Chapter 6.5, 20