COURSE 1

Transaction Management and Distributed Databases

Assessment / Other Details

- Final grade
  - 50% - laboratory activity / practical test
  - 50% - written exam

- Course details (*bibliography, course slides, seminars, lab descriptions etc*)

http://www.cs.ubbcluj.ro/~tzutzu

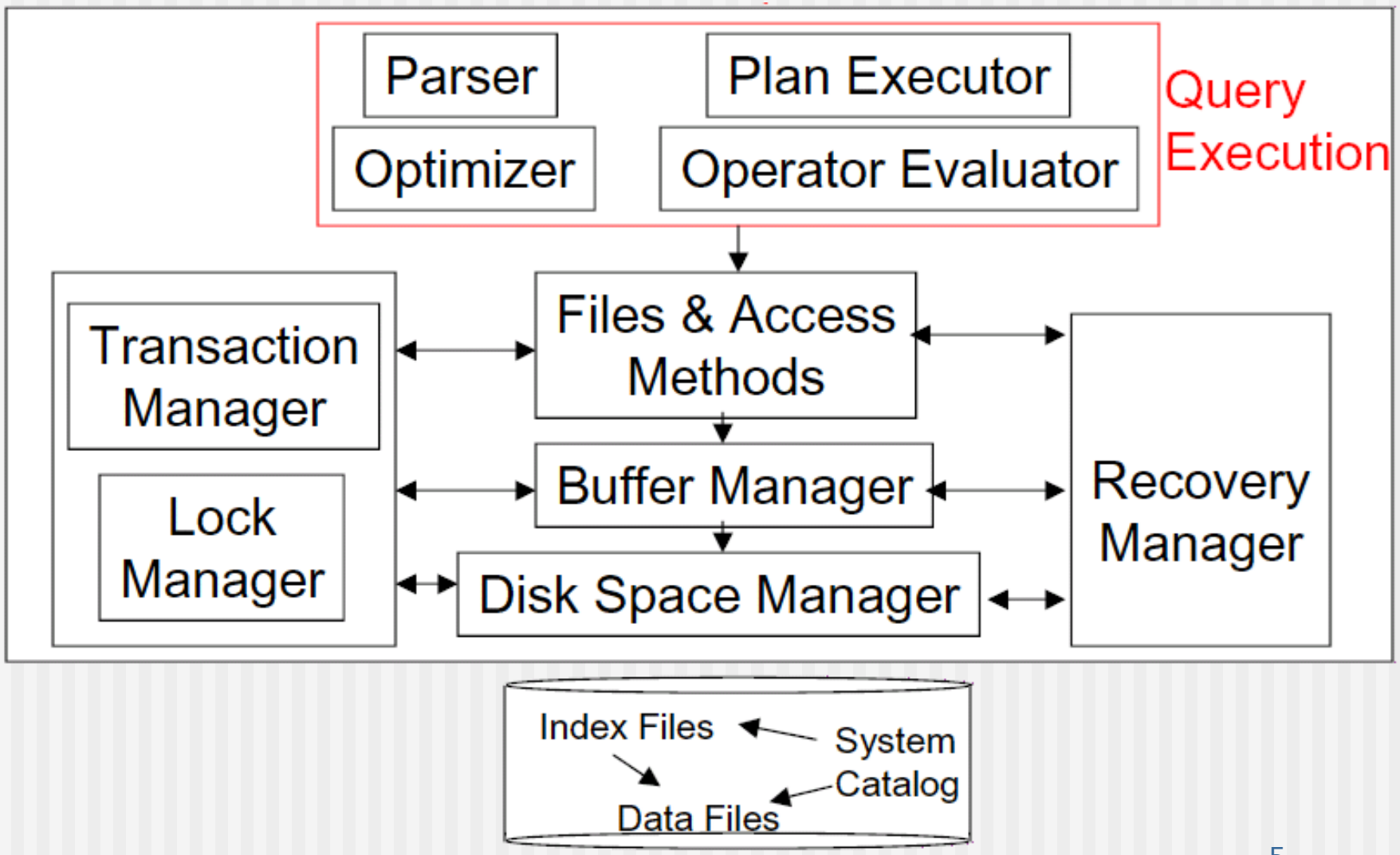
Laboratory

- Working “environments”
  - DBMS: MS SQL Server
  - App. Development: .NET / C#
  - Data access: ADO.NET

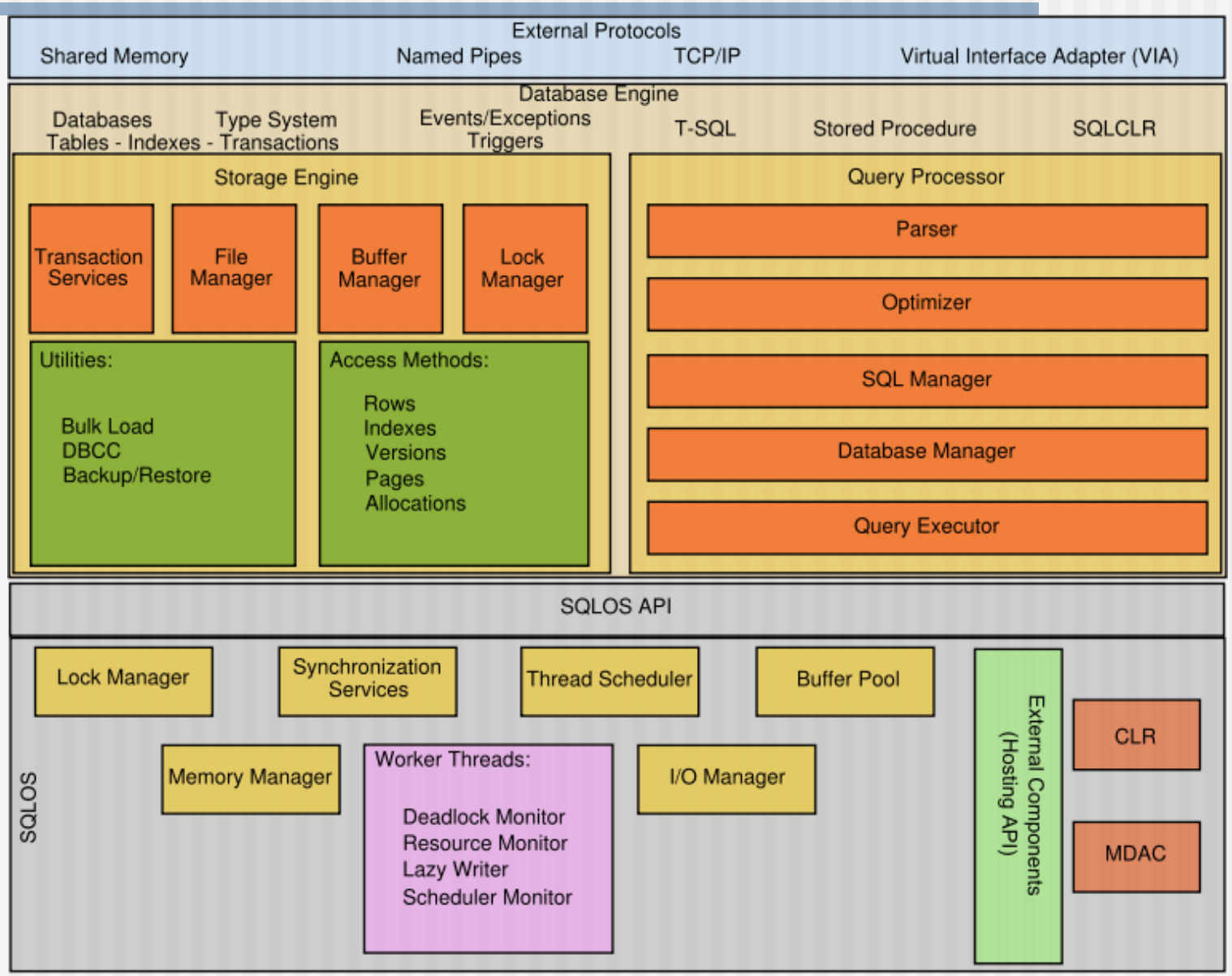
Course Brief Presentation

- Transaction Management
- Concurrency Control
- Crash Recovery
- External Sorting
- Evaluation of Relational Operations
- Query Optimization
- Distributed / Parallel Databases
- Security
- Data Mining / Data Warehousing

Detailed Structure of a DBMS

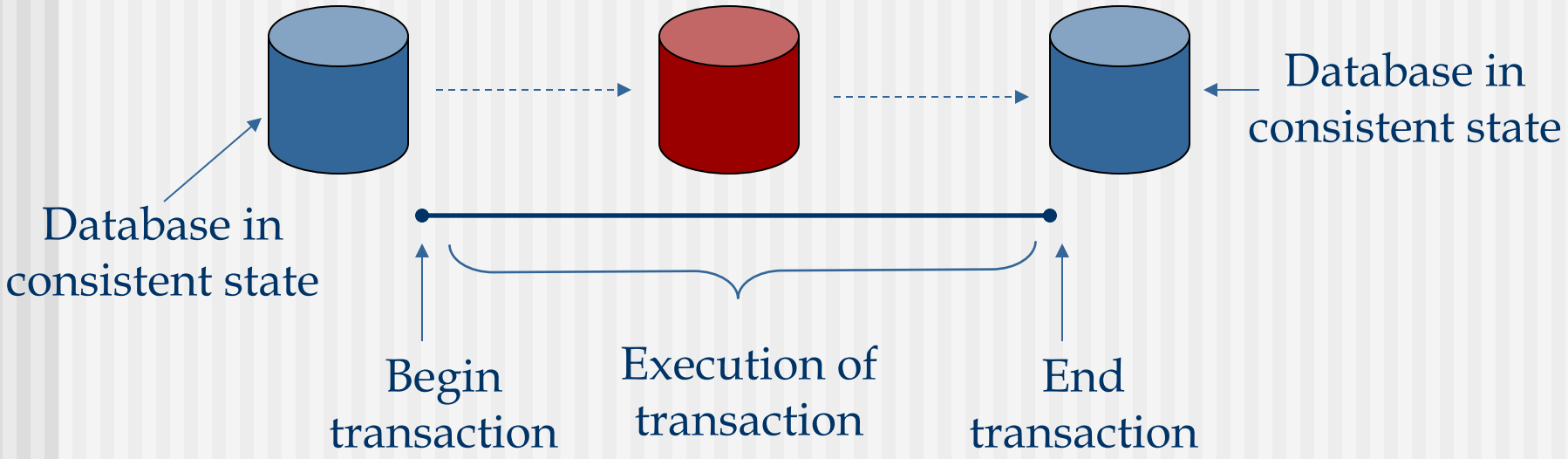


MS SQL Server structure



Transactions

- A *transaction* is a sequence of operations which performs one "single logical function" on a shared database.
- A *transaction* makes consistent transformations of system states while preserving system consistency.



Transactions (cont.)

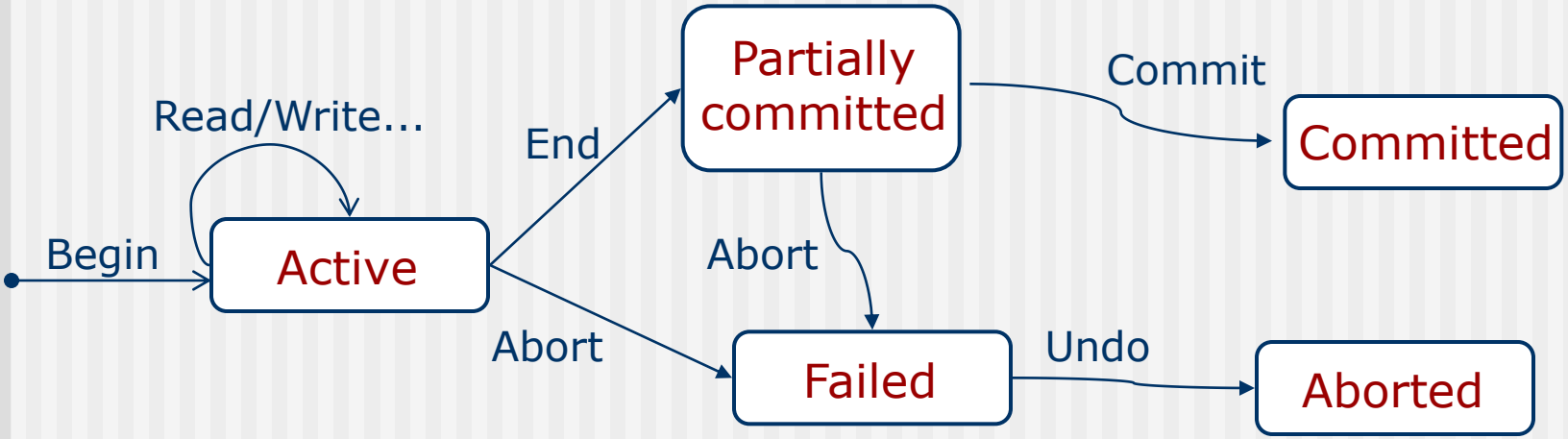
- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about Read/Write.
- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.

Transaction Operations

- A database transaction is the execution of a program that include database access operations:
  - Begin – transaction
  - Read
  - Write
  - End – transaction
  - Commit – transaction
  - Abort – transaction
  - Undo
  - Redo

State of Transactions

- Active: the transaction is executing
- Partially Committed: the transaction ends after execution of final statement
- Committed: after successful completion checks
- Failed: the normal execution can no longer proceed
- Aborted: after the transaction has been rolled back



Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues*: Effect of *interleaving* transactions and *crashes*.

Transaction Properties - **ACID**

Atomicity (all or nothing)

- A transaction is *atomic*: transaction always executing all its actions in one step, or not executing any actions at all.

Consistency (no violation of integrity constraints)

- A transaction must preserve the consistency of a database after execution. (responsibility of the user)

Isolation (concurrent changes invisible →serializable)

- Transaction is protected from the effects of concurrently scheduling other transactions.

Durability (committed updates persist)

- The effect of a committed transaction should persist even after a crash.

Atomicity

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A user can think of a transaction as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
- The activity of ensuring atomicity in the presence of system crashes is called *crash recovery*.

Consistency

- A transaction which executes *alone* against a consistent database leaves it in a *consistent* state.
- Transactions do not violate database integrity constraints.
- Transactions are *correct* programs

Isolation

- If several transactions are executed concurrently, the results must be the same as if they were executed serially in some order (serializability).
- An incomplete transaction cannot reveal its results to other transactions before its commitment.
  - Necessary to avoid cascading aborts.

Durability

- Once a transaction commits, the system must guarantee that the result of its operations will never be lost, in spite of subsequent failures.
- Database recovery

Example

- Consider two transactions:

  T1: BEGIN A=A+100, B=B-100 END
  T2: BEGIN A=1.06*A, B=1.06*B END

- Intuitively, the first transaction is transferring 100€ from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Example (Contd.)

Consider a possible interleaving (*schedule*):

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

This is OK. But what about:

T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*B

The DBMS's view of the second schedule:

T1: R(A), W(A), R(B), W(B)
T2: R(A), W(A), R(B), W(B)

Conflicts of Operations

- If two transactions only read a data object, they do not conflict and the order is not important
- If two transactions either read or write completely separate data objects, they do not conflict and the order is not important.
- If one transaction writes a data object and another either reads or writes the same data object, the order of execution is important.
  - WR conflict: T2 reads a data objects previously written by T1
  - RW conflict: T2 writes a data object previously read by T1
  - WW conflict: T2 writes a data object previously written by T1

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

T1: R(A), W(A), R(B), W(B), **A**
T2: R(A), W(A), **C**

- Unrepeatable Reads (RW Conflicts):

T1: R(A), R(A), W(A), **C**
T2: R(A), W(A), **C**

- Overwriting Uncommitted Data (WW Conflicts, "blind writes"):

T1: W(A), W(B), **C**
T2: W(A), W(B), **C**

Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency:
  - System automatically inserts lock/unlock requests and schedules actions of different transactions in such a way as to ensure that the resulting execution is equivalent to executing the transactions one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - *Consistent state*: Only the effects of committed transactions seen.