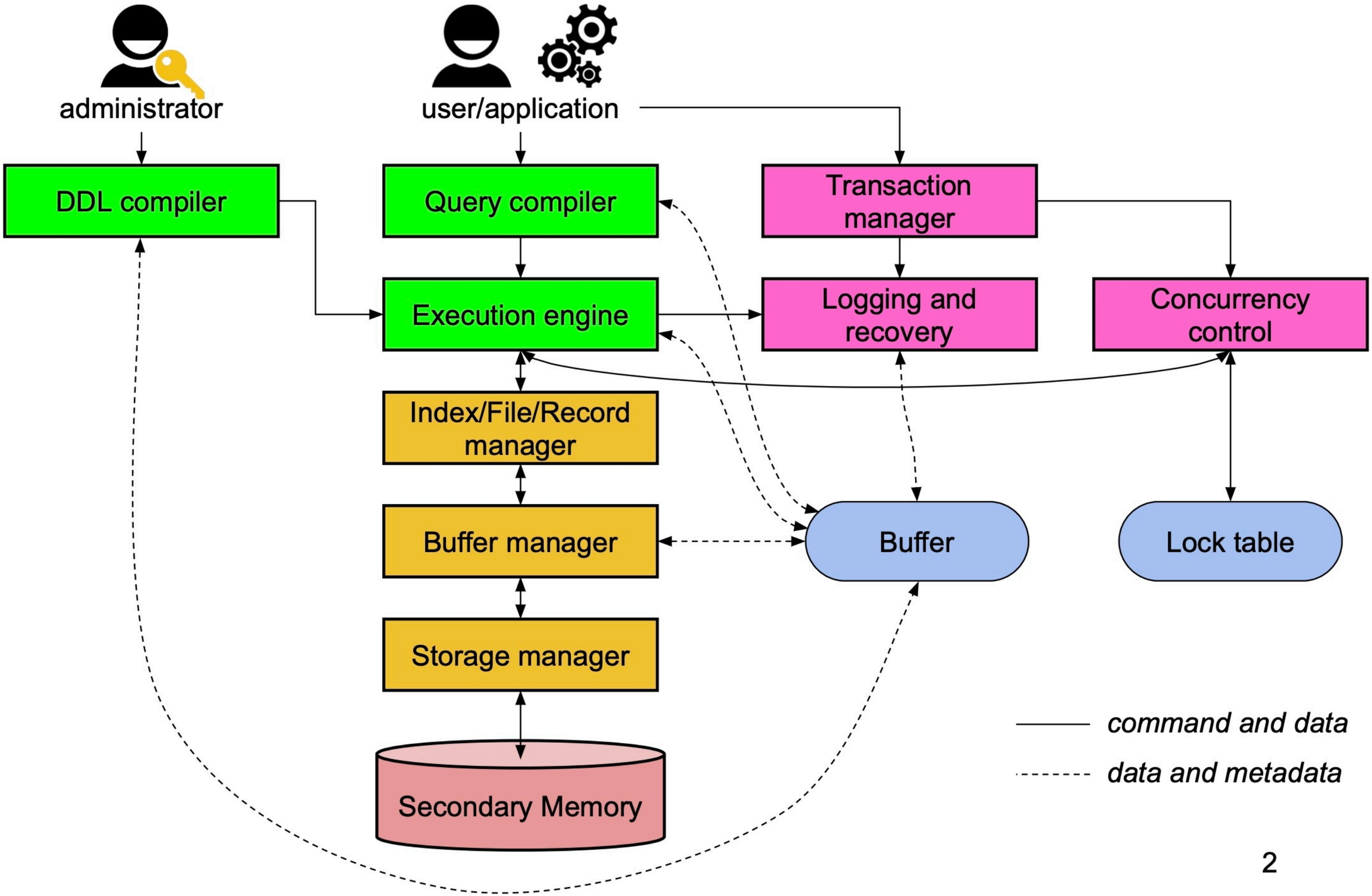
Databases
Lab

Transactions and Concurrency Control

Flavio Bertini

flavio.bertini@smartdata.cs.unibo.it

Relational DBMS Architecture



DBMS and User Programs

- A DBMS is a multi-user environments (e.g., bank, company). Each user accesses and/or modifies the data through a program in a **high-level query language** (e.g., SQL).

- A program is a sequence of:
  - **read(X)**: reads a DB item named X;
  - **write(X)**: writes a value into the DB item named X.

- To simplify, a **transaction** represents the execution of **a user program** in a DBMS that includes **one or more read/write** operations.

- Main problems:
  - **Concurrent execution of transactions.**
  - **Crash recovery.**

Transaction - an example

- Suppose a bank employee transfers $500 from A's account (n°42177) to B's account (n°12202).

```
start transaction;
    update BankAccount Set Balance = Balance + 500
           Where AccountNumber = 12202;
    update BankAccount Set Balance = Balance - 500
           Where AccountNumber = 42177;
end transaction;
```

- What if:
  - At the same time, other users wants to transfer money also → **concurrent execution**.
  - Network failure → **crash recovery**.

DBMS Concurrency: Motivations

- Disk accesses are slow and CPU should not wait.
- For performance reasons, **multiple transactions** can be performed **in a concurrent way**, keeping the CPU always busy.

- **Problem**:
  - The user must **not have any perception of interleaving execution**, that is the multiplicity of transactions being executed.

DBMS Crash Recovery: Motivations

- Due to an unforeseen event, **a transaction may not end** as expected.
- **Problem**:
  - The DBMS must **ensure that other transactions are not affected** by the incorrect execution of that transaction and must leave the database in a consistent state (e.g., bank transfer).

ACID Properties of Transactions

- To ensure **concurrent and safe execution of transactions** and **crash recovery**, each transaction must have the following properties:
  - **ATOMICITY**: a transaction is an atomic unit, that is either all of its operations are executed or none. There must be no state in a database where a transaction is left partially completed
  - **CONSISTENCY**: all transactions are expected to preserve the consistency of the database, that is constraints or expectations about relationships.
  - **ISOLATION**: each transaction must appear to be executed as if no other transaction is executing at the same time.
  - **DURABILITY**: once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Consistency and Isolation

- The **consistency** must be ensured **by the user** implementing that transaction.
- The **isolation**, in relation to other competing transactions, is guaranteed by making sure that the outcome of the scheduling is the same **as the sequential execution** of transactions.
- No matter whether the **execution of the read and write** operations forming the transactions **are interleaved**.
- However, the **DBMS does not guarantee** that transactions are executed in **a specific order**.

Atomicity and Durability

- Transaction abort cases:
  1. **Internal anomaly** due to execution. The DBMS stops such transaction.
  2. **Exception conditions** detected by the transaction, which is self-suspended.
  3. **System crash**, for instance, due to a hardware, software or network error.

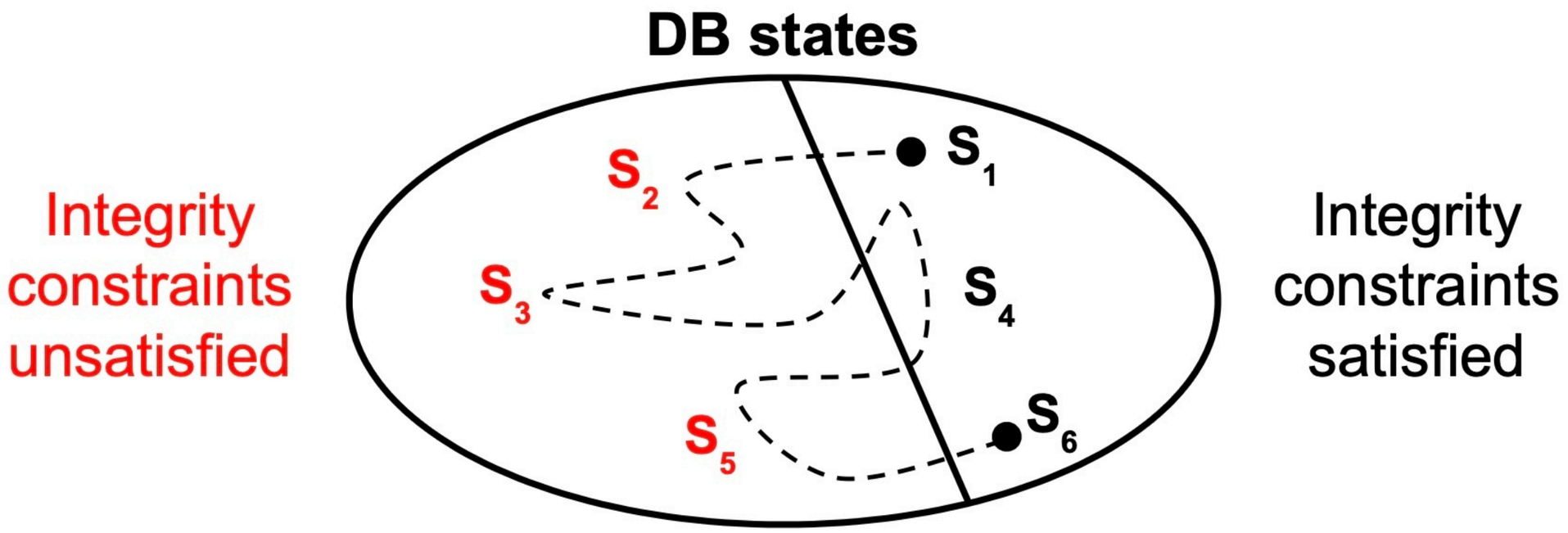
- An aborted transaction could leave the DB in an inconsistent state.

- The DBMS guarantees the atomicity through a **log file of all the operations** that have been made.

- If an abort occurs, **the DBMS checks the log and rolls back the DB** to the last consistent state prior to the abort of the transaction. **Log files also guarantee durability.**

Consistency and Transactions

- *Assumption*: the developer has defined some **integrity constraints** to be preserved (e.g., “The wages’ sum has to be less than the available budget.”).
- A transaction:
  - **always starts in a consistent state** where all the integrity constraints are satisfied;
  - **may pass through** intermediate **inconsistent states**;
  - **always terminates in a consistent state** where all the integrity constraints are satisfied.



Schedule

- **A transaction is a sequence of read and write** operations on database objects (e.g., attribute values, tuples, etc.).
- Each transaction must specify its final action:
  - **commit** (transaction successfully completed) or …
  - **abort** (termination and undo of actions performed)

- A **schedule S is a sequence of actions** (read, write, commit and abort) taken **from a set of transactions**.
- **The order of actions in S is the same** as in the corresponding transactions.
- A **schedule represents a chronological execution** sequence of the actions present in the transactions involved.

Schedule - an example

- Given the following two transactions on A, B and C (database objects):

  T_1: $read_1(A)$, $write_1(A)$, $read_1(C)$, $write_1(C)$
  T_2: $read_2(B)$, $write_2(B)$

- **A possible schedule S** is as follows:

  S: $read_1(A)$, $write_1(A)$, $read_2(B)$, $write_2(B)$, $read_1(C)$, $write_1(C)$

- Schedules allow describing the intermittent executions of competing transactions.
  - **Complete schedule** - it includes commit or abort operation for each involved transaction.
  - **Serial schedule** - for each transaction, all the operations are executed consecutively (i.e., only one transaction is active at a time).

Serializable Schedule

A serializable schedule is a schedule whose **effect** on any consistent database instance **is guaranteed to be identical** to that of some **complete serial schedule**.

Serializable Schedule - an example (1)

SCHEDULE	Transaction		DB objects	
	T_1	T_2	A=50	B=50
		read(A)		
		A=A*2		
		write(A)	100	
		read(B)		
		B=B*2		
		write(B)		100
	read(A)			
	A=A+100			
	write(A)		**200**	
	read(B)			
	B=B+100			
	write(B)			**200**

S_1 is a serial schedule: transactions are executed consequently, no matter the order.

Serializable Schedule - an example (2)

SCHEDULE	Transaction		DB objects	
	T_1	T_2	A=50	B=50
	read(A)			
	A=A+100			
	write(A)		150	
	read(B)			
	B=B+100			
	write(B)			150
		read(A)		
		A=A*2		
		write(A)	**300**	
		read(B)		
		B=B*2		
		write(B)		**300**

S_2 is a serial schedule: transactions are executed consequently, no matter the order.

Serializable Schedule - an example (3)

SCHEDULE	Transaction		DB objects	
	T_1	T_2	A=50	B=50
	read(A)			
	A=A+100			
	write(A)		150	
		read(A)		
		A=A*2		
		write(A)	**300**	
	read(B)			
	B=B+100			
	write(B)			150
		read(B)		
		B=B*2		
		write(B)		**300**

S_3 is serializable but not a serial schedule: its effect on the database state is the same as that of some serial schedule.

Serializable Schedule - an example (4)

SCHEDULE	Transaction		DB objects	
	T_1	T_2	A=50	B=50
	read(A)			
	A=A+100			
	write(A)		150	
		read(A)		
		A=A*2		
		write(A)	**300**	
		read(B)		
		B=B*2		
		write(B)		100
	read(B)			
	B=B+100			
	write(B)			**200**

S_4 is not serializable.

Interleaved Execution Anomalies

- An interleaved execution of **two consistent and committed transactions** (e.g., T_1 and T_2) can leave the DB in an inconsistent state. We can have three types of conflicts:
  - **Write-Read Conflict** (a.k.a. **dirty read**): T_2 **reads** an object **written** by T_1, but T_1 **hasn't committed** yet.
  - **Read-Write Conflict** (a.k.a. **unrepeatable reads**): T_2 **writes** an object **read** by T_1, but T_1 **hasn't committed** yet.
  - **Write-Write Conflict** (a.k.a. **lost update**): T_2 **writes** an object **written** by T_1, but T_1 **hasn't committed** yet.

S_4 presents a write-read conflict.

Write-Read Conflict

- Given the following two transactions T_1 and T_2 and the schedule S:

  T_1: A := A + 100, B := B - 100

  T_2: A := A * 1.06, B := B * 1.06

- **Please note**: T_1 writes a value in A that could make the DB inconsistent, and this value is read by T_2.
- **Problem**: A is written by T_1 and read by T_2 before T_1 commits.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
	read(B)
	write(B)
	commit
read(B)	
write(B)	
commit	

Read-Write Conflict

- Given the following two transactions T_1 and T_2 and the schedule S:

  T_1: A := A + 1

  T_2: A := A - 1

T_1	T_2
read(A)	
	read(A)
	write(A)
	commit
write(A)	
commit	

- **Please note**: if T_1 repeats the reading of A, it will get a different value.
- **Problem**: A is written by T_2 before T_1 commits.

Write-Write Conflict

- Given the following two transactions T_1 and T_2 and the schedule S:

  T_1: A := 1000, B := 1000

  T_2: A := 2000, B := 2000

T_1	T_2
write(A)	
	write(A)
	write(B)
	commit
write(B)	
commit	

- **Please note**: the values of A and B are equal at the end of each of the two transactions.
- **Problem**: A and B have different values at the end of the schedule.

Phantom Anomalies

- This problem occurs when a transaction reads an object (e.g., a table) at two different times and it finds that the object is no longer the same.

T1	T2
read(X)	
	read(X)
insert/delete(a, X)	
	read(X)

Aborted Transactions

- In principle, all actions of **aborted transactions must be cancelled**. However, this is **impossible in some cases**.

- **Problem**: T_2 reads the value of A which shouldn't have read, due to a write-read conflict.

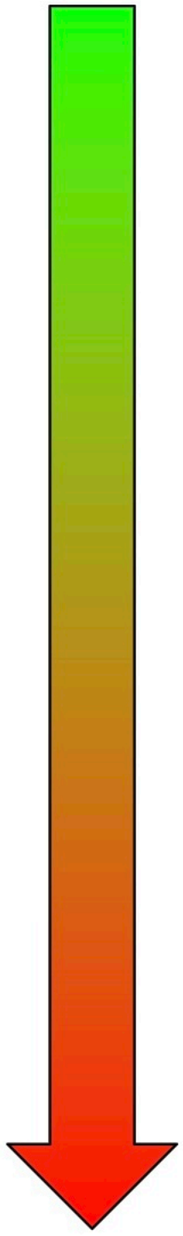
- **Unsatisfactory solution**: T_2 should also be aborted, but it would violate the property of durability of transactions (ACI**D**).

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
	read(B)
	write(B)
	commit
read(B)	
write(B)	
abort	

Transaction Parameters

- **ACCESS MODE** sets the permission to changes tables used in the transaction:
  - **READ ONLY** allows reading the DB only, attempting to make changes to the database causes an error.
  - **READ WRITE** allows reading and writing operations.
- **STATEMENT MODE** specifies the action to be taken when a transaction ends.
- **ISOLATION LEVEL** specifies how to handle transactions that modify the database, ranging from low to high isolation level:
  - **READ UNCOMMITTED**
  - **READ COMMITTED**
  - **REPEATABLE READS**
  - **SERIALIZABLE**

Isolation Level



- **READ UNCOMMITTED** - The transaction **requires locks for writing** objects but no locks for reading. The transaction may read not yet committed changes made by other transactions.
- **READ COMMITTED** - The transaction **requires locks for writing** and **shared locks for reading**. It guarantees that any data read is committed at the moment it is read.
- **REPEATABLE READS** - The transaction **requires locks for reading and writing any DB object** and releases them when it commits only. No table-level locks are allowed to scan it.
- **SERIALIZABLE** - As the previous one, including **table-level locks for scanning**.

Isolation Level vs Conflicts

Level	Dirty read (Write-Read)	Lost update (Read-Write)	Unrepeatable read (Write-Write)	Phantom
READ UNCOMMITTED	may occur	may occur	may occur	may occur
READ COMMITTED	don't occur	may occur	may occur	may occur
REPEATABLE READS	don't occur	don't occur	don't occur	may occur
SERIALIZABLE	don't occur	don't occur	don't occur	don't occur

Concurrency and Interleaving

- Interleaving is necessary and desirable to improve the performances, however, **not all the possible schedules are feasible**.
- **Some actions must be rolled back** when transactions are aborted.
- **Question**: what checks should be made to avoid conflicts?

Concurrency Control Approaches

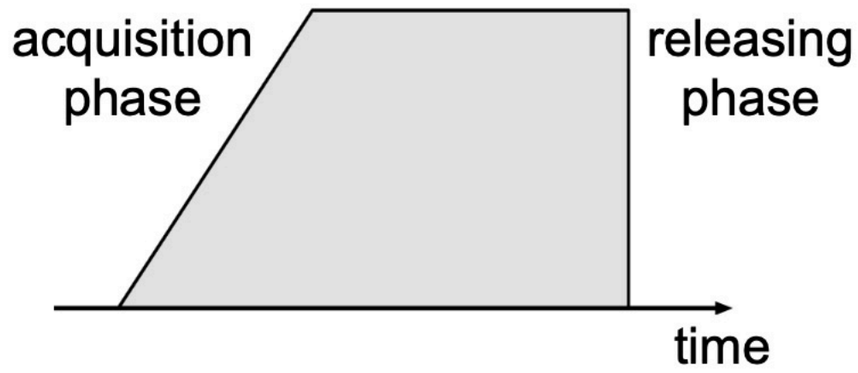
- There are different approaches:
  - RESTRICTIVE - Look for **serializable scheduling** that avoids conflicts **through data locking protocols** (conflict-serializability).
  - OPTIMISTIC - Run all the transactions concurrently, **checking conflicts before committing**.
  - TIMESTAMPING - Assigning **timestamps to transactions** that have either written or read objects, and compare these values **to determine the order** of the operations in the scheduling.

Restrictive Concurrency Control

- The DBMS must ensure the **absence of conflicts** and the **rollback of aborted transactions**.
- **Strict Two-phase Locking** (Strict 2PL):
  - If a transaction wants to **read/write an object**, it has to **require exclusive access** to it.
  - **After releasing a lock, a transaction cannot require new ones.**
  - The transaction **releases** all of its **exclusive accesses** when it **commits**.
- **Strict 2PL guarantees serializability**, in particular:
  - If transactions access **different objects**, they could be **freely interleaved**.
  - Otherwise, if at least two transactions want to access the **same object** and at least one transaction wants to modify it, they have to be **run serially**.
- The **Lock manager tracks**, for each object within the DB, the **exclusive accesses in a lock table**.

Strict 2-Phase Locking - an example

- Exclusive access in detail:
  - **Shared lock** of A for reading: S(A).
  - **eXclusive lock** of A for writing: X(A).



- Rules of the game:
  - If a transaction wants to read A, it needs first a shared lock.
  - If a transaction wants to write A, it needs first an exclusive lock.
  - A shared lock blocks exclusive lock request from other transactions but does not block any shared lock requests from other transactions.
  - A exclusive lock blocks any shared/exclusive lock requests.
  - All locks are released after commit.

- T_1: S(A), R(A), X(A), W(A), X(B), R(B), W(B), commit
- T_2: X(A), R(A), W(A), X(B), R(B), W(B), commit

Deadlock Prevention

- A locking-based concurrency control can cause **deadlocks** that must be **avoided or resolved by the DBMS**.

- DBMS usually assigns **a priority to each transaction** depending on the start time (i.e., timestamp).

- If T_i requires a lock owned by T_j, the lock manager could implement one of these two policies:

  - **Wait-Die**: if T_i is the older transaction, T_i waits. Otherwise, T_i is killed and restarted later with random delay but with the same timestamp.

  - **Wound-Wait**: if T_i is the older transaction, T_j is killed and restarted later with random delay but with the same timestamp. Otherwise, T_i waits.

Deadlock Detection and Resolution

- If **deadlocks are rare**, a DBMS **let deadlock occur and solve** them when they occur instead of adopting a policy to avoid them.

- The DBMS can adopt one of the two most common approaches:
  - The lock manager maintains a structure called **waits-for graph** that it uses to identify **deadlock cycles**. The graph is periodical analysed and deadlock cycles **are solved by aborting some transactions**.
  - If a **transaction waits for a period longer than a given timeout**, the lock manager assumes that the transaction **is deadlocked and aborts it**.

Optimistic Concurrency Control

- The locking-based protocol adopts a pessimistic approach to preventing conflicts from occurring.
- The optimistic approach assumes that transactions do not conflict (or rarely conflict).
- **Validation** is an optimistic approach where transactions are **allowed to access data without locks**, **and** at the appropriate time, it **checks** that the transactions have behaved in a serializable manner.
- Transaction are executed in three phases:
  - **Read**: the transaction is executed by reading the data from the DBMS and writing in a private area.
  - **Validation**: Before the commit, the DBMS checks that no conflicts have occurred. If so, it aborts the transaction and restarts it automatically.
  - **Write**: If the validation phase is successful, the data written in the private area is copied into the DBMS.

Timestamping Concurrency Control

- **Timestamping** is another optimistic concurrency control approach that **assigns each transaction the timestamp** TS of its start time.
- For each operation a_i run by T_i:
  - If the operation a_i conflicts with the operation a_j run by T_j and **$TS_i < TS_j$**, then **a_i must be performed before a_j**.
  - If an operation run by **T violates this order**, the transaction T **is aborted and restarted with a greater TS**.

Crash Recovery

- A DBMS **Logging and recovery manager** must ensure:
  - **Atomicity**: operations performed by **non-committing transactions are rolled back**.
  - **Persistency**: operations performed by committing transaction must **survive to system crashes**.

Advanced Recovery and Integrated Extraction System

- Advanced Recovery and Integrated Extraction System (**ARIES**) is a recovery algorithm which is run by the logging and recovery manager on system crashes.
- ARIES is used by IBM DB2, Microsoft SQL Server and many other DBMS. There are **three phases**:
  - **Analysis phase**: it identifies the **dirty pages** of the buffer pool (i.e., the changes not yet written to the disk) and the **active transactions** at the time of the crash.
  - **Redo phase**: starting from a given **checkpoint** in the log file, it **repeats all the operations** and takes the DB back to the state it was in at the time of the crash.
  - **Undo phase**: **cancels the operations** of transactions that were active at the time of the crash but **weren't committed** in the reverse order.

ARIES Principles

- **Write-Ahead Logging** - Any changes to a DB object must be first recorded in the log file. Then the log file must be written on secondary memory. Finally, the corresponding modified pages can be updated.
- **Repeating History During Redo** - On restart after a crash, the system is brought back to the state that it was in before the crash. The operations of the transactions still active at crash time are cancelled.
- **Logging Changes During Undo** - Changes made to the DB while undoing transactions are logged to ensure such an action isn't repeated in the event of repeated restarts/crashes.

Log File

- The log file **tracks all the actions** performed by the DBMS. It is physically **organized in records** stored in stable storage, that is assumed to survive crashes and hardware failures.
- Log records are sequentially ordered with a unique id, that is the **Log Sequence Number** (LSN). The LSN is monotonically increasing and allows to identify/access records.
- The **most recent part** of the log file, so-called log tail, **is kept in log buffers** and saved periodically in stable storage. The log file and data are written to disk with the same mechanisms.

Logging Data Structures

- The **Dirty Page Table** keeps a record of all the **pages ID** that have been modified and not yet written back in stable storage, and the first LSN that caused that page to become dirty.

- The **Transaction Table** keeps a record of all **transactions ID** that are currently running, and the LSN of the last log entry they caused.

Log File Record

<LSN, Transaction ID, Page ID, Redo, Undo, Previous LSN>

- The Transaction ID and Page ID fields identify the transaction and the page, respectively.
- The Redo and Undo fields keeps information about the changes the log record saves and how to undo them.
- The Previous LSN field is a reference to the previous log record that was created for the same transaction. It allows to roll back aborted transaction easily.

Creation of a Log File Record

- A record is written to the log file for each of these events:
  - **Page Update**: a record must be written in stable storage before actually changing the page data. It keeps both the old and the new page value to make possible undo and redo operations.
  - **Commit**: a record tracks that a transaction has completed successfully, and the log tail is written in stable storage (#log tail pages << #data pages).
  - **Abort**: a record tracks an aborted transaction, and the transaction undo is started.
  - **End**: After commit/abort, some finalization operations are needed at the end of which an end record is written.
  - **Undoing Operation**: During a recovery or while undoing the operations, a special kind of log record is written, the Compensation Log Record (CLR). A CLR record is never reverted and tracks that an operation has already been undone.

Checkpoint

- A checkpoint is a **snapshot of the DB state**. Checkpoints **reduce recovery time**. Instead of having to run through the whole log file, it is just necessary to run backwards until a checkpoint is found.

- ARIES creates checkpoints in three steps:
  - A **begin-checkpoint** record is written to the log file.
  - An **end-checkpoint** record, containing both the Dirty Page Table and the Transaction Table, is written on the log file.
  - After the end-checkpoint record is written in stable storage, a **Master Record containing the LSN of the begin-checkpoint** record is written to a known place in stable storage.

- This type of checkpoint is called **fuzzy checkpoint** and is **inexpensive in terms of performances**. It does not interrupt the normal operation of the DBMS and does not require writing the pages of the buffer pool.

ARIES Crash Recovery

- **Analysis phase**
  - Determine the **location of the log file** from which to **start the redo phase**, that is the beginning or the last checkpoint.
  - Determine which buffer pool pages contain **modified data that weren't written** yet at the time of the crash.
  - Identify the **transactions were running** at the time of the crash.

- **Redo phase**
  - From the Dirty Page Table, ARIES identify the **minimal LSN** of a dirty page. From there, it **starts redoing the actions until the crash**, in case they weren't persisted already.

- **Undo phase**
  - The changes of uncommitted transactions have to be undone to restore the database to a consistent state.