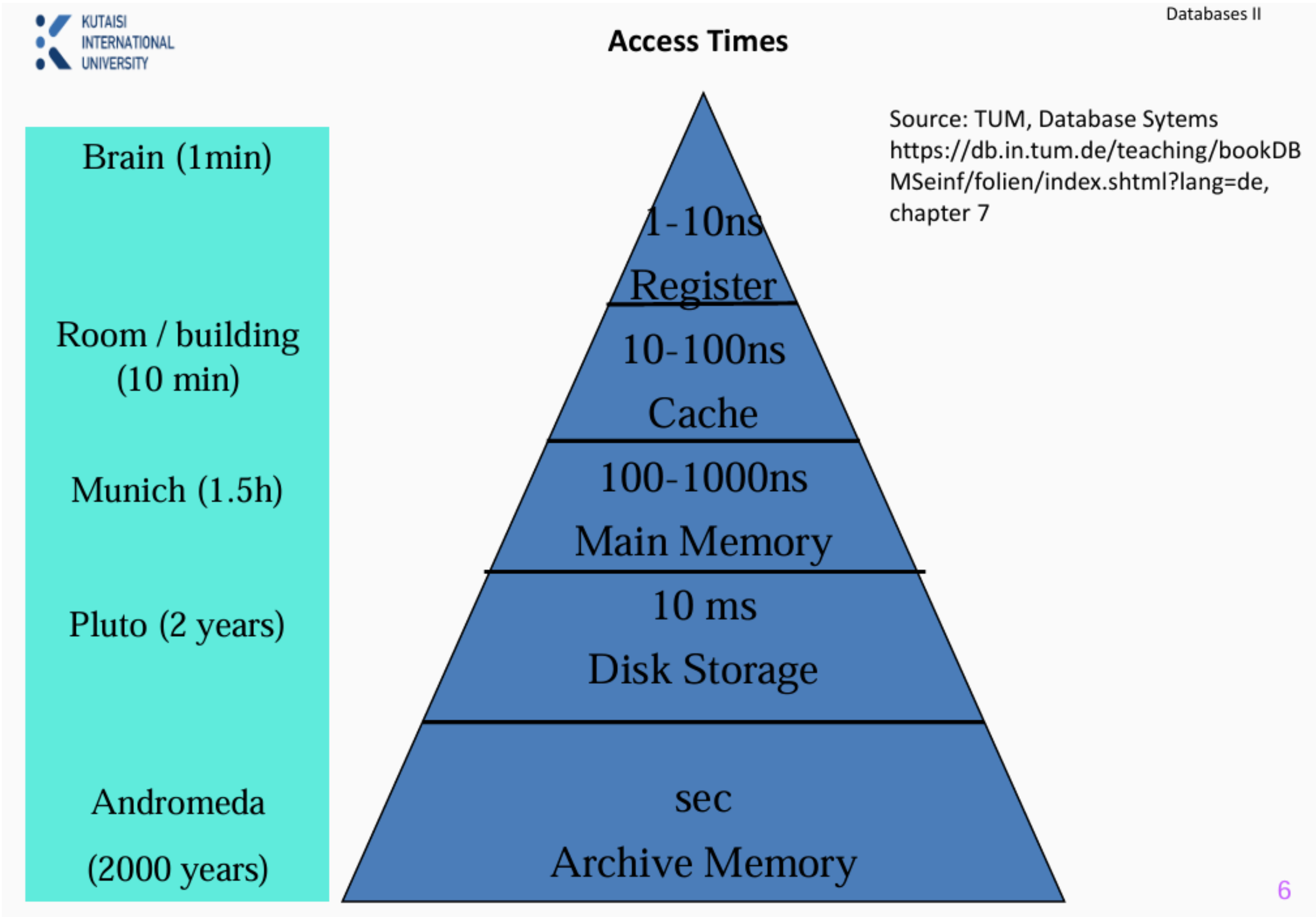
Lecture 3

Type	Lecture

Access times



Shows how long it takes to access different memory types:

- Fastest is **Registers (1–10ns)**
- Slowest is **Archive/Backup (seconds)**
- Disk access is slow compared to RAM, which is why we avoid writing to disk too often (WAL helps with this).

Write-Ahead-Log (WAL)

WAL ensures atomicity and durability efficiently. (WAL is a page in buffer).

- Before actual data is changed on disk, a log of the change is written (what you change, operations are written in the WAL file, basically update, delete insert).
- If the system crashes, WAL can replay those logs to restore the correct state.
- Only the WAL file is flushed to disk on `COMMIT`, not the entire database.

Periodically, system writes data/index pages to disk, and syncs them with WAL logs, basically deleting them from the logs, so that WAL does not grow forever and it makes crash recovery faster.

Described Anomalies and Isolation Levels

Isolation Levels	SQL Standard Anomalies				Serialization Anomalies	
	Dirty Write	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update	Write Skew
Read Uncommitted **prevents**	Yes	No	No	No	No	No
Read Committed **prevents**	Yes	Yes	No	No	No	No
Repeatable Read **prevents**	Yes	Yes	Yes	No	Depends on Implementation	No
Serializable **prevents**	Yes	Yes	Yes	Yes	Yes	Yes

Anomalies and Isolation Levels PostgreSQL

Isolation Levels	SQL Standard Anomalies				Serialization Anomalies	
	Dirty Write	Dirty Read	Non-Repeatable Read	Phantom Read	Lost Update	Write Skew
Read Committed **prevents**	Yes	Yes	No	No	No	No
Repeatable Read **prevents**	Yes	Yes	Yes	**Yes**	**Yes**	No
Serializable **prevents**	Yes	Yes	Yes	Yes	Yes	Yes

Dirty write

A transaction overwrites data that another transaction has written, but not yet committed. All ACID implementations prevent dirty writes.

▼ Example

PostgreSQL prevents this. If T1 tries to update a row that T2 modified but hasn’t committed, T1 must wait.

Dirty Read

A transaction reads another transaction's writes before they have been committed. Reading uncommitted data.

▼ Example

T1 sees a temporary value from T2, which later rolls back. Now T1 has read a value that never officially existed.

Lost update

Two transactions concurrently perform a read-modify-write cycle. One transaction updates data (one or more rows) and commits. The second transaction updates the same data and overwrites the committed update without noticing / respecting the update of the first transaction. Update of the first committed transaction is lost.

update of same data concurrently and t1 overwrites changes of t2 without respecting modifications

Write Skew Anomaly

Two transactions (T1 and T2) running in parallel, each reads the current state and makes a decision based on that, but their writes violates the initial assumption. they work on different rows, but take the same row as a reference.

can be prevented if

- transactions run in serial execution mode, one after the other.
- transactions run in a mode equivalent to serial execution mode.

▼ Example

1	T1	T2	Result
2	Begin		
3	n_on_duty = select count(*) from personnel where on_duty=TRUE	Begin	T1 returns 101 personnel on duty
4		n_on_duty = select count(*) from personnel where on_duty=TRUE	T2 returns 101 personnel on duty
5	if (n_on_duty > 100) (update personnel set on_duty = False where name= "Alice")		
6	commit;		100 personnel are on duty
7		if (n_on_duty > 100) (update personnel set on_duty = False where name= "Bob")	
8		Commit;	99 personnel are on duty

difference between a Lost Update and a Write Skew Anomaly

Lost update

Two transactions update the same data, but one update is lost.

Write skew

Two transactions update **related** data in a way that creates an inconsistent state (for example violates a constraint).

Non-repeatable read

A transaction sees different states of the database (different values for one object) at different points in time. **occurs when a transaction reads the same row twice but gets different data each time**. For example, suppose transaction 1 reads a row. Transaction 2 updates or deletes that row and commits the update or delete.

▼ Example

KUTAISI INTERNATIONAL UNIVERSITY

Databases II

Non-Repeatable Read Anomaly

Example:
T1: Returns the student names where s_balance < 5
T2 Adds 5 to s_balance of student

1	T1	T2	Result
2	Begin		
3	Select s_name where s_balance < 5	Begin	T1 returns the names of 10 students, among them student x
5		Update student set s_balance = s_balance + 5 where s_name = x	T2 **updates** s_balance of student x, s-balance of x now > 5
6		**commit**	
7	Select s_name where s_balance < 5		T1 returns the names of 9 students, student x is not among them
8	Commit		

Phantom read

A **phantom read** occurs when a transaction **re-executes a query and sees new rows** inserted (or deleted) by another transaction, even though its own snapshot remains unchanged.

▼ Example

KUTAISI INTERNATIONAL UNIVERSITY

Databases II

Phantom Read Anomaly

T1: Counts the students where s_balance < 5
T2 Inserts a new student with default s_balance = 0

1	T1	T2	Result
2	BoT		
3	Count (*) where s_balance < 5;	BoT	T1 returns x
4		Insert student	T2 **inserts** a new student
6		commit	
7	Count (*) where s_balance < 5;		T1 returns x+1, a "phantom row"
8	Commit		

For the application this is a similar problem as non-repeatable read.
For the database, preventing phantom reads is differnt from preventing non-repeatable reads. Any idea, why?

Different because it can not lock the row that doesn't exist yet

Difference between non-repeatable read and phantom reads

Non-repeatable read	Phantom reads
happen when existing rows are updated or deleted.	happen when new rows are inserted or deleted in a range-based query.

Concurrency Control Protocols

Two main types:

- **2PL (Two-Phase Locking)** – pessimistic, used in older systems.
- **MVCC (Multiversion Concurrency Control)** – optimistic, used in PostgreSQL, Oracle. MVCC allows transactions to work on **snapshots** of data rather than locks.

Lock-Based CC Protocols

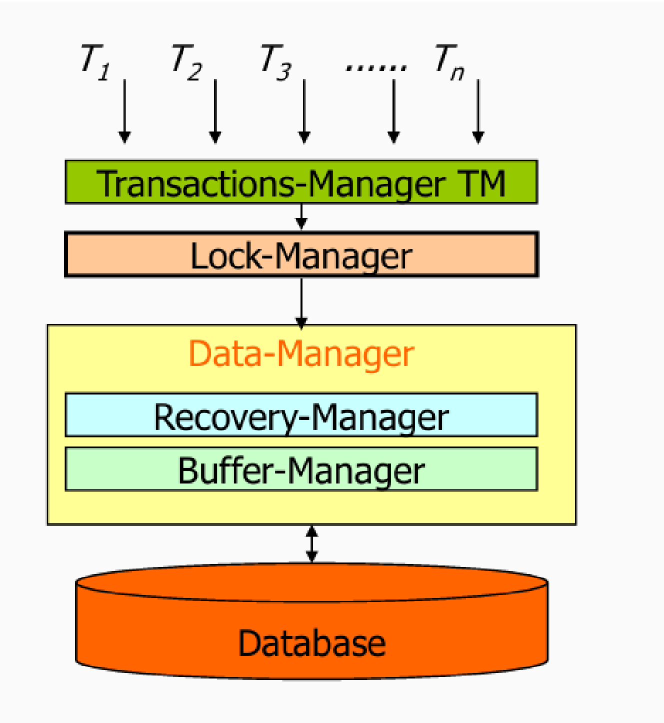
Two types of locks:

- **Shared (S-lock)**: multiple reads allowed. Its purpose is to read a resource, multiple transactions can hold an S-lock on the same resource at the same time. It allows read and prevents writing. you use it when you want to read a value, but you don't want it to be changed while you're reading it.
- **Exclusive (X-lock)**: only one write allowed. Its purpose is to write a resource, Only one transaction can hold an X-lock on a resource. It allows both reading and writing and prevents other transactions from reading or writing the same resource. you use it when you want to update a row and you need to ensure no one else even reads it until you're done.

Compatibility

	S-lock Requested	X-lock Requested
S (already held)	✅ (ok)	❌ (must wait)
X (already held)	❌ (must wait)	❌ (must wait)

Architecture



Components involved in concurrency:

- **Lock Manager**: tracks who locked what.
- **Recovery Manager**: handles crashes.
- **Buffer Manager**: manages memory pages.
- **Transaction Manager**: coordinates all of this.