Introduction to Database Systems IDBS - Autumn 2024

Lecture 10 - Transactions in RDBMs

ACID Properties
Logging
Locking

Readings: PDBM 14

Omar Shahbaz Khan

Wake Up Task!

- Course Evaluation on LearnIT

- Take 15 min.
- What do you like about the course and what can be done better in the future?

-- TODO

- Transactions
    - ACID Properties
    - Buffer Management
    - Logging
    - Recovery
    - Locking

-- TODO

- Transactions
  - ACID Properties
  - Buffer Management
  - Logging
  - Recovery
  - Locking

Transactions

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

- Assume that account 12345 starts as ‘savings’ with balance=500.
- What are the possible balance values after running transactions A, B and C?

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

Assume that account 12345 starts as 'savings' with balance=500

A→B→C: -490	A→C→B: -490	B→A→C: -495
B→C→A: -535	C→A→B: -510	C→B→A: -535

Consider the following transactions on the relation accounts(no, balance, type):

Transaction A

```
UPDATE accounts SET balance=balance*1.02 WHERE type='savings';
UPDATE accounts SET balance=balance*1.01 WHERE type='salary' AND balance > 0;
UPDATE accounts SET balance=balance*1.07 WHERE type='salary' AND balance < 0;
```

Transaction B

```
UPDATE accounts SET type='salary' WHERE no=12345;
```

Transaction C

```
UPDATE accounts SET balance=balance-1000 WHERE no=12345;
```

Assume that account 12345 starts as 'savings' with balance=500 and they are interleaving

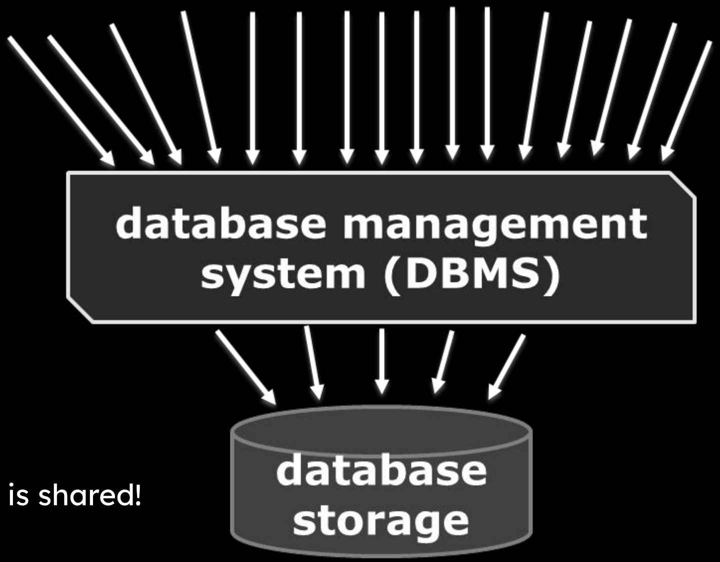
A.1: 510
B: 510 (salary)
A.2: 515.10
C: -484.90
A.3: -518.843

Why do we need Transactions?

- Transactions group a set of database operations
  - Atomic, all operations succeed or none do
- May require shared database resources (rows, columns, tables)
  - Can lead to conflicts with multiple concurrent transactions
- Can be run sequentially or concurrently
  - Trade-off between consistency (data integrity) and performance (latency)

Why do we need Transactions?

Many concurrent requests



- from different applications
- from several users in the same application

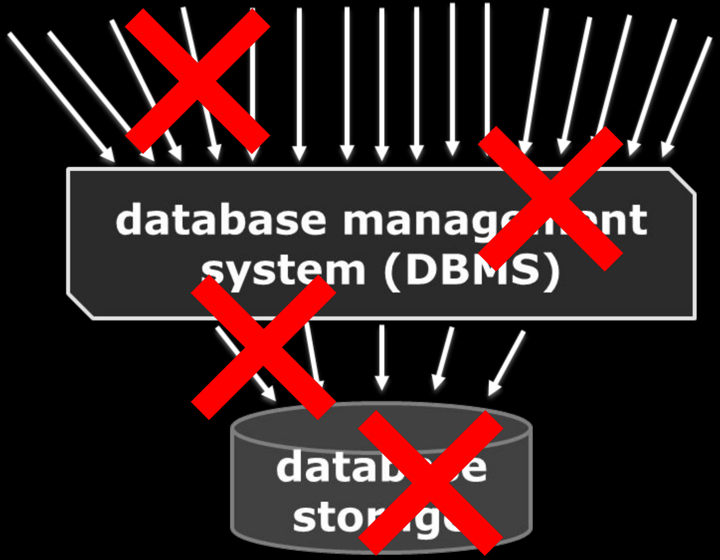
Data is shared!

DBMS must ensure reliable operations over shared data despite many concurrent accesses

Why do we need Transactions?

ANYTHING CAN CRASH!

Many concurrent requests



DBMS must ensure reliable operations over shared data despite many concurrent accesses

To the end user everything is fine, thanks to transactions

Recovery	Constraints, Triggers	Concurrency Control	Recovery
ATOMICITY	**CONSISTENCY**	**ISOLATION**	**DURABILITY**
A Transaction is "one operation"	Each Transaction moves the DB from one consistent state to another	Each Transaction is alone in the world	Persistence of successful transactions even through system failure

ACID Properties of Transactions

- Atomicity: Each transaction runs to completion or has no effect at all.
- Consistency: After a transaction completes, the integrity constraints are satisfied.
- Isolation: Transactions executed in parallel have the same effect as if they were executed sequentially.
- Durability: The effect of a committed transaction remains in the database even if the system crashes.

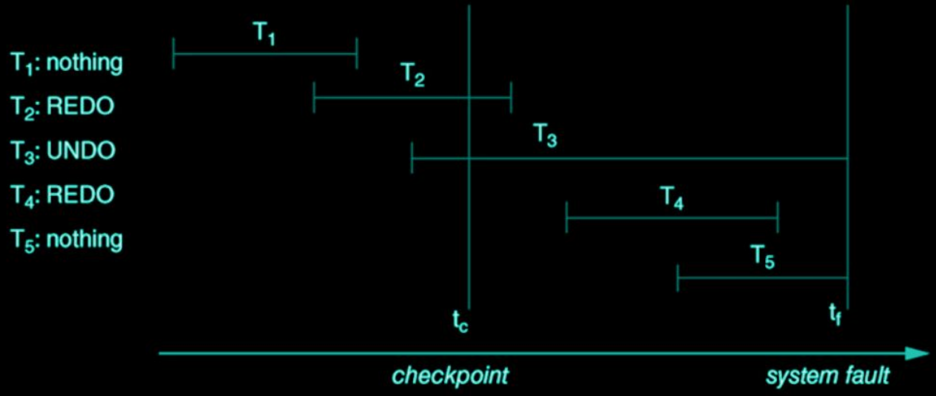
How to Implement Transactions?

- Consistency ~= satisfying constraints
  - Use indexes for primary and foreign key, triggers, …
- Atomicity and Durability = tracking changes
  - Logging “before” values to UNDO changes
  - Logging “after” values to REDO changes
- Isolation = preventing corrupting changes
  - Pessimistic: Locking to prevent conflicts
  - Optimistic: Time stamps to detect conflicts

Atomicity and Durability Issues

- Atomicity
  - Transactions abort
  - Systems crash
- Durability
  - Systems crash

- Upon restart
  - Want to <u>see</u> effects of T1, T2, T4
  - Want to <u>remove</u> the effects of T3, T5



Buffer Management (RAM v Disk)

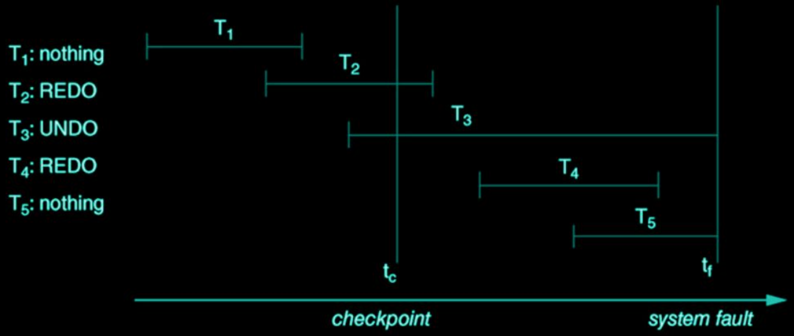
- Once a transaction completes, it performs a COMMIT
- A COMMIT ≠ changes are on disk
- Buffer Management Policies determine when changes are moved from RAM to disk
  - The transaction log is always updated / written to disk
  - FORCE / NO FORCE policies affect data pages of committed transactions
  - STEAL / NO STEAL policies affect data pages of uncommitted transactions

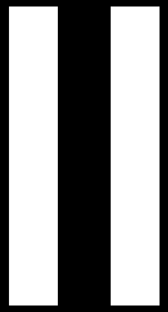
FORCE / NO STEAL

- FORCE = write changed pages to disk at COMMIT
  - Book: “immediate update” policy
  - Ensures Durability (assuming writes are atomic)
  - Increases response time
  - Will we FORCE changes to disk at COMMIT? NO
- NO STEAL = allow updated pages to be replaced
  - Book: NO STEAL = “deferred update” policy
  - Ensures Atomicity (can simply discard at abort)
  - Increases response time
  - Will we guarantee NO STEAL of dirty pages? NO

NO FORCE / STEAL

- NO FORCE: Changes in RAM after COMMIT
  - What if system crashes?
  - Need to remember the new value of P to be able to REDO the changes
- STEAL: Changes to disk before COMMIT
  - What if transaction aborts? system crashes?
  - Need to remember the old value of P to be able to UNDO the changes





Write-Ahead Logging (WAL Protocol)

- Write-Ahead Logging
  1. Before any changes are written to disk
     we force the corresponding log record to disk
  2. Before a transaction is committed
     we force all log records for the transaction to disk

- #1 ensures Atomicity
- #2 ensures Durability

Key Concept: The Log

- Write REDO and UNDO info to log
  - Ordered list of REDO/UNDO info
  - Think of an infinite file with append only!

- Log processing must be fast – why? and how?
  - Write minimal info to log (diff)
    - <xid, pageID, offset, length, old_data, new_data> + control info
    - Many log entries per page
  - Ensure sequential writes!
    - Writing a log entry ⇨ writing all previous entries
    - <u>**Put log on its own disk!**</u>

Transactions in MMDBMSs

- ACI…D
  - Are we ok with losing all data?
  - How do we make changes persistent?
    - Transaction Log on Disk
    - Snapshots on Disk (Copy of the database state)

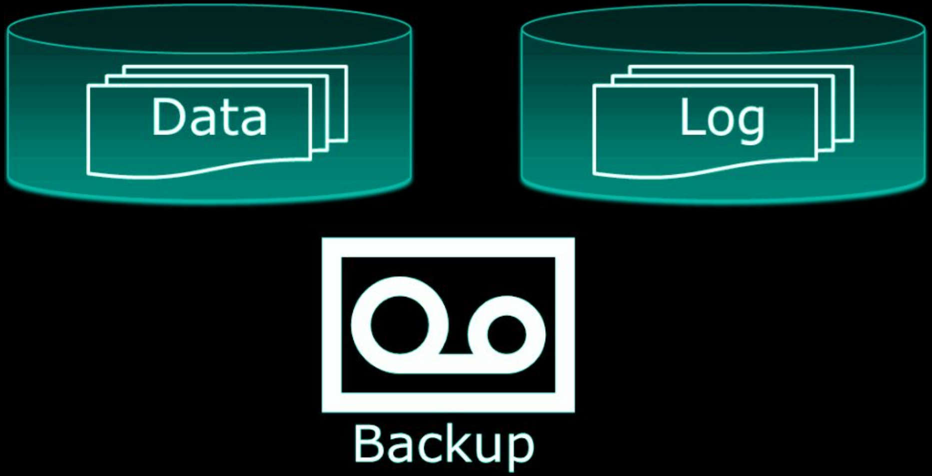
Restart Recovery

1. Analyze information about transactions from the last checkpoint

2. <u>REDO</u> the changes of committed transactions that did not make it to disk

3. <u>UNDO</u> the changes of uncommitted transactions that accidentally made it to disk

Database Setup for DBAs



- On crash:
  - Data + Log = Recovery succeeds
  - Backup + Log = Recovery succeeds
  - Data + Backup (No Log) = Recovery FAILS!

https://www.visualpharm.com/free-icons/set/hard-drive

High Availability

- Typical approach: Second (failover) server
  - Takes over in case the primary fails
- Transaction log used to keep it up to date

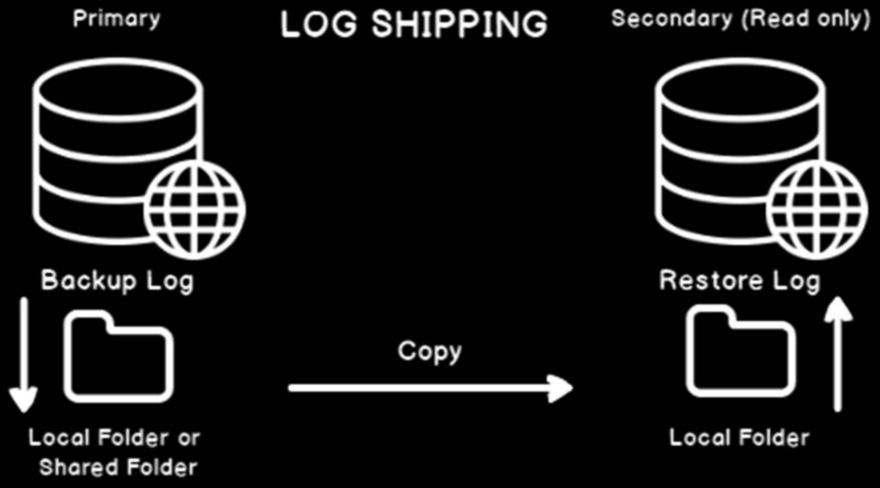


Image source: https://www.sqlshack.com/sql-server-transaction-log-and-high-availability-solutions/

How to Implement Transactions?

- Consistency ~= satisfying constraints
  - Use indexes for primary and foreign keys, triggers, …

- Atomicity and Durability = tracking changes
  - Logging “before” values to undo changes
  - Logging “after” values to redo changes

- Isolation = preventing corrupting changes
  - **Pessimistic: Locking to prevent conflicts**
  - Optimistic: Time stamps to detect conflicts

Isolation and Serializability

- Want transactions to satisfy *serializability*:
  - The state of the database should always look as if the committed transactions ran in some *serial schedule*

- The scheduler of the DBMS is allowed to choose the order of transactions:
  - It is not necessarily the transaction that is started first, which is first in the serial schedule

A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order

- Problems:
  - Transactions must *wait* for each other, even if unrelated (e.g. requesting data on different disks)
    - Some transactions may take very long, e.g. when external input or remote data is needed during the transaction
  - Possibly smaller throughput (Why?)

A Simple Scheduler

- A simple scheduler would maintain a queue of transactions and carry them out in order
- Some believe this is fine for *transaction processing*, especially for MMDBs

The End of an Architectural Era
(It's Time for a Complete Rewrite)

Michael Stonebraker
Samuel Madden
Daniel J. Abadi
Stavros Harizopoulos

Nabil Hachem
AvantGarde Consulting, LLC
nhachem@agdba.com

Pat Helland
Microsoft Corporation
phelland@microsoft.com

Interleaving Schedulers

- Most DBMSs still have schedulers that allow the actions of transactions to interleave
- However, the result should be **as if** some serial schedule was used
  - A non-serial schedule that yields the same outcome as a serial schedule is known as a serializable schedule
- We will now study a mechanism that *enforces* “serializability”: **Locking**
- Other methods exist: Time stamping / optimistic concurrency control
  - Out of scope for this course

Locks

- In its simplest form, a lock is a right to perform operations on a database element
- Only one transaction may hold a lock on an element at any time
- Locks must be requested by transactions and granted by the locking scheduler
- Typically, two types of locks: Read/Shared or Write/Exclusive

Two-Phase Locking

- Growing Phase: Acquire locks
  - When reading a database resource, get a shared (S) lock
  - When writing a database resource, get an exclusive (X) lock

- Shrinking Phase: No new locks can be acquired, and start releasing locks

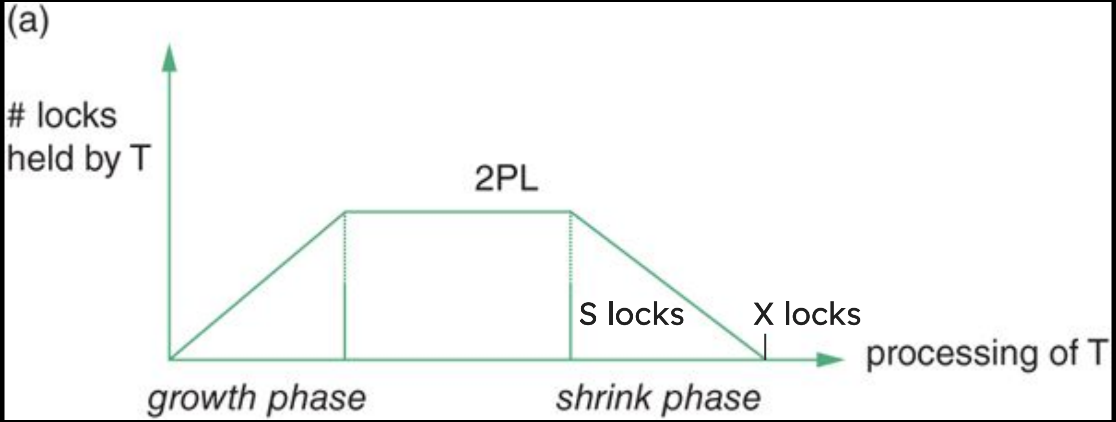
- Multiple variations:
  - Static 2PL: All locks acquired before running the transaction
  - Strict 2PL: Release only shared locks during shrinking phase
  - Rigorous 2PL: Release locks at COMMIT/ABORT

- Commonly implemented, since:
  - Simple to understand and works well in practice
  - It makes transaction rollback easier to implement

- But: Optimistic approaches are gaining in popularity

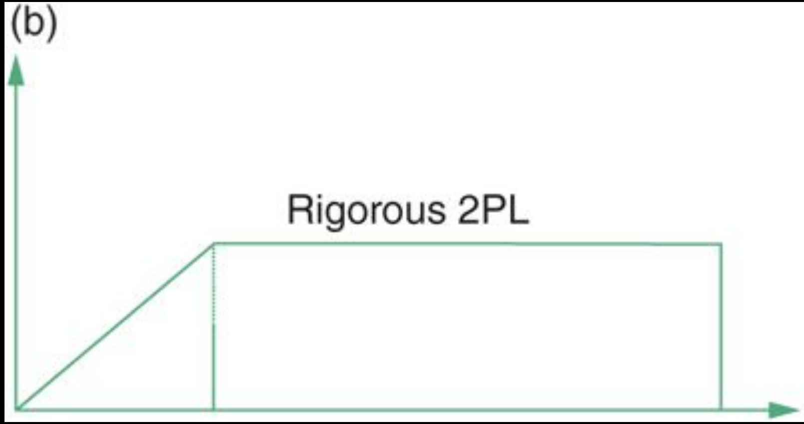
Strict Two-Phase Locking

- Strict 2PL protocol:
  1. Before reading a record/page, get a shared (S) lock
     Before writing a record/page, get an exclusive (X) lock
  2. A record/page cannot have an X lock at the same time as any other lock
  3. Release S locks during shrinking phase and X locks on COMMIT/ABORT



Rigorous Two-Phase Locking

- Rigorous 2PL protocol:
  1. Before reading a record/page, get a shared (S) lock
     Before writing a record/page, get an exclusive (X) lock
  2. A record/page cannot have an X lock at the same time as any other lock
  3. Release all locks on COMMIT/ABORT



Locks and Deadlocks

- The DBMS sometimes must make a transaction wait for another transaction to release a lock.
- This can lead to deadlock if e.g. A waits for B, and B waits for A.
- In general, we have a deadlock exactly when there is a cycle in the waits-for graph.
- Deadlocks are resolved by aborting some transaction involved in the cycle.

Avoiding Deadlocks

- Upgrade requests can also deadlock

SELECT :x=counter FROM table WHERE *<condition>*	UPDATE table SET counter = counter+1 WHERE *<condition>*	SELECT FOR UPDATE :x=counter FROM table WHERE *<condition>*
:x=:x+1		:x=:x+1
UPDATE table SET counter = :x WHERE *<condition>*	SELECT :x=counter FROM table WHERE *<condition>*	UPDATE table SET counter = :x WHERE *<condition>*

- Order Matters:
  - With consistent order of access, deadlocks are avoided
  - Why do B+ tree accesses not deadlock? - Same traversal order
  - Optimizer may not allow control over order!

Phantom Tuples

- Suppose we lock tuples where A=1 in a relation, and subsequently another tuple with A=1 is inserted.

- For some transactions this may result in unserializable behaviour, i.e., it will be clear that the tuple was inserted during the course of a transaction.

- Such tuples are called phantoms.

Phantom Example

```
CREATE TABLE B (x INTEGER, y INTEGER);
INSERT INTO B VALUES (1, 2);


BEGIN;
SELECT MIN(y) FROM B WHERE x = 1;
                        BEGIN;
                        INSERT INTO B VALUES (1, 1);
                        COMMIT;

-- Repeat the SAME read!
SELECT MIN(y) FROM B WHERE x = 1;
COMMIT;
```

Avoiding Phantoms

- Phantoms can be avoided by putting an exclusive lock on a relation before adding tuples.
  - However, this leads to poor concurrency.

- A technique called “index locking” can be used to prevent other transactions from inserting phantom tuples but allow most non-phantom insertions.

- In SQL, the programmer may choose to either allow phantoms in a transaction or insist they should not occur.

Isolation Levels in Modern Systems

- READ UNCOMMITTED
  a transaction can read uncommitted changes
- READ COMMITTED
  a transaction only reads committed data,
  some other transaction may overwrite this data
- REPEATABLE READ
  a transaction only reads committed data,
  other transactions cannot overwrite this data,
  but phantoms are possible
- SERIALIZABLE
  ensures serializable schedule with no anomalies

May violate the "I" in ACID

Ensures "I" in ACID

Database	Default Isolation	Maximum Isolation
Actian Ingres 10.0/10S	S	S
Aerospike	RC	RC
Akiban Persistit	SI	SI
Clustrix CLX 4100	RR	?
Greenplum 4.1	RC	S
IBM DB2 10 for z/OS	CS	S
IBM Informix 11.50	Depends	RR
MySQL 5.6	RR	S
MemSQL 1b	RC	RC
MS SQL Server 2012	RC	S
NuoDB	CR	CR
Oracle 11g	RC	SI
Oracle Berkeley DB	S	S
Oracle Berkeley DB JE	RR	S
Postgres 9.2.2	RC	S
SAP HANA	RC	SI
ScaleDB 1.02	RC	RC
VoltDB	S	S
Legend	*RC: read committed, RR: repeatable read, S: serializability, SI: snapshot isolation, CS: cursor stability, CR: consistent read*	

The entire world doesn't run on ACID!

But an important part of it does!

Figure taken from: [When is "ACID" ACID? Rarely]

-- TODO -> DONE

- ✓ Transactions
  - ✓ ACID Properties
  - ✓ Buffer Management
  - ✓ Logging
  - ✓ Recovery
  - ✓ Locking

Takeaways

You're banished to
the Phantom Zone

ACID:

- Atomicity:
  - All or nothing
  - COMMIT/ABORT/ROLLBACK
  - Write Ahead Logging (WAL)
- Consistency:
  - Constraints
  -
- Isolation:
  - Locking (Pessimistic)
  - Isolation levels
- Durability:
  - Write Ahead Logging (WAL)
  - Backups and recovery protocols

Next Time in IDBS…

Introduction to Database Systems IDBS - Autumn 2024

Lecture 11 - Scale and Cloud

Scaling-Out
NoSQL
Eventual Consistency
CAP Theorem

Readings: PDBM 11

Omar Shahbaz Khan