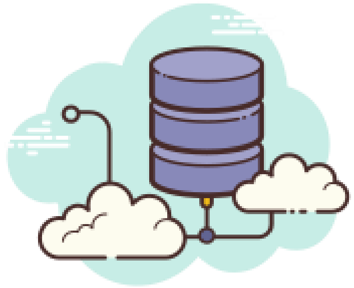
INF115 Lecture 12: *Transactions*

Adriaan Ludl
Department of Informatics
University of Bergen

Spring Semester 2021

Chapter 10: *Transactions*



Learning Goals:

- **Understand** *what* ***transactions*** *are*

  and which ***properties*** *they should have.*

- Use SQL to **confirm** and **cancel** *transactions.*

- Understand the **challenges** that **error** situations and **concurrency** present for the *system* with regard to *transactions*.

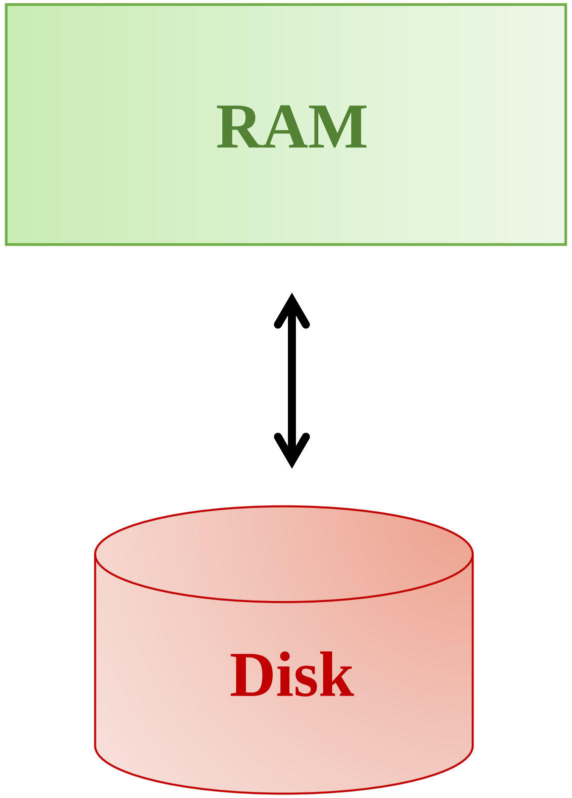
- How to use **logging** to handle *error* situations.

- How to use **locks** to deal with *concurrency* problems.

Snapshot of a database

- Data is **stored** on disk, but must be read into the RAM for **calculation**.
  - At a **particular** given **moment**, a part of the database will exist **only in RAM**.

- Database systems are in **production** for a **long time**.
  - Must handle **error / failure situations**.



What is an «operation» ?

SQL statements can handle **many rows**:

```
UPDATE Ansatt
SET Lønn = Lønn * 1.1
```

- Salaries for all employees are **updated**.
- One SQL statement can thus perform many «***operations***».

Conversely, we may want to consider **multiple SQL statements** as one "**composite**" operation.

- Example from Hobbyhuset:
- New orders require the insertion of rows in the *Order* and *OrderLine* tables, as well as an update of the *Product* (*Vare*).

Transactions

A **transaction** is a logical *operation* on the database.

- It may involve one or more tables.
- It may consist of one or more SQL statements.

Transactions can be closed in two ways:

- `COMMIT` : confirm, changes are made permanent.
- **ROLLBACK** : undo, the transaction is "rolled back“.

Challenges:

- **Errors** such as disk crashes and power outages.
- **Simultaneous users** which can hinder for each other.

Transactions in SQL

Employee 22 receives an order for 5 units of product 4034 from customer 1003 – and creates order 2020:

```
START TRANSACTION;
INSERT INTO Ordre(OrdreNr,KNr,AnsNr)
     VALUES (2020,1003,22);
INSERT INTO Ordrelinje(OrdreNr,VNr,Antall)
     VALUES (2020,'4034',5);
UPDATE Vare
SET Antall = Antall-5
WHERE VareNr='4034';
COMMIT;
```

- After the first / the first two insertions (INSERT)

  the database is **not** in a **valid** state.

Transactions in MySQL

Auto-commit is standard. It can be turned off with

```
SET autocommit = 0;
```

Or an **explicit** start with START TRANSACTION:

```
START TRANSACTION
INSERT INTO Ordre(OrdreNr,KNr,AnsNr)
VALUES (2020,1003,22);

-- more commands ...
COMMIT;
```

Transactions in MySQL

Auto-commit is standard. It can be turned off with

```
SET autocommit = 0;
```

Or an **explicit** start with START TRANSACTION:

```
START TRANSACTION
INSERT INTO Ordre(OrdreNr,KNr,AnsNr)
VALUES (2020,1003,22);

-- more commands ...
COMMIT;
```

ACID Properties

Atomicity: All or none of the sub-operations of a transaction must be completed.

Consistency: A transaction moves the database from one **valid** state to another **valid** state.

Isolation: The effect of transactions in **progress** should not be observable by other transactions.

Durability: The effect of **completed** (comitted) transactions is **stored** in the database and shall not be lost due to errors.

ACID Properties

Atomicity: All or none of the sub-operations of a transaction must be completed.

Consistency: A transaction moves the database from one **valid** state to another **valid** state.

Isolation: The effect of transactions in **progress** should not be observable by other transactions.

Durability: The effect of **completed** (comitted) transactions is **stored** in the database and shall not be lost due to errors.

ACID Properties

Atomicity: All or none of the sub-operations of a transaction must be completed.

Consistency: A transaction moves the database from one **valid** state to another **valid** state.

Isolation: The effect of transactions in **progress** should not be observable by other transactions.

Durability: The effect of **completed** (comitted) transactions is **stored** in the database and shall not be lost due to errors.

ACID Properties

Atomicity: All or none of the sub-operations of a transaction must be completed.

Consistency: A transaction moves the database from one **valid** state to another **valid** state.

Isolation: The effect of transactions in **progress** should not be observable by other transactions.

Durability: The effect of **completed** (comitted) transactions is **stored** in the database and shall not be lost due to errors.

The Transaction Log

TransID	Table	RowID	Attribute	Before	After
101	*** BEGINTRANS				
101	VARE	102375	ANTALL	112	113
101	VARE	102542	ANTALL	56	66
101	*** COMMIT				

- Everything is registered in the log before the database is updated.
- Once COMMIT is written to the log, the decision has been made.
- The log can be used to update the database.
- ROLLBACK: Any updates to the database will be "rolled back".
- The log is also used for recovery after errors.

The Transaction Log

TransID	Table	RowID	Attribute	Before	After
101	*** BEGINTRANS				
101	VARE	102375	ANTALL	112	113
101	VARE	102542	ANTALL	56	66
101	*** COMMIT				

- Everything is registered in the log before the database is updated.
  - Once COMMIT is written to the log, the decision has been made.
- The **log** can be used **to update the database**.
  - ROLLBACK: Any updates to the database will be "rolled back".
- The **log** is also used **for recovery after errors**.

Quizz on *Transactions* (part 1)

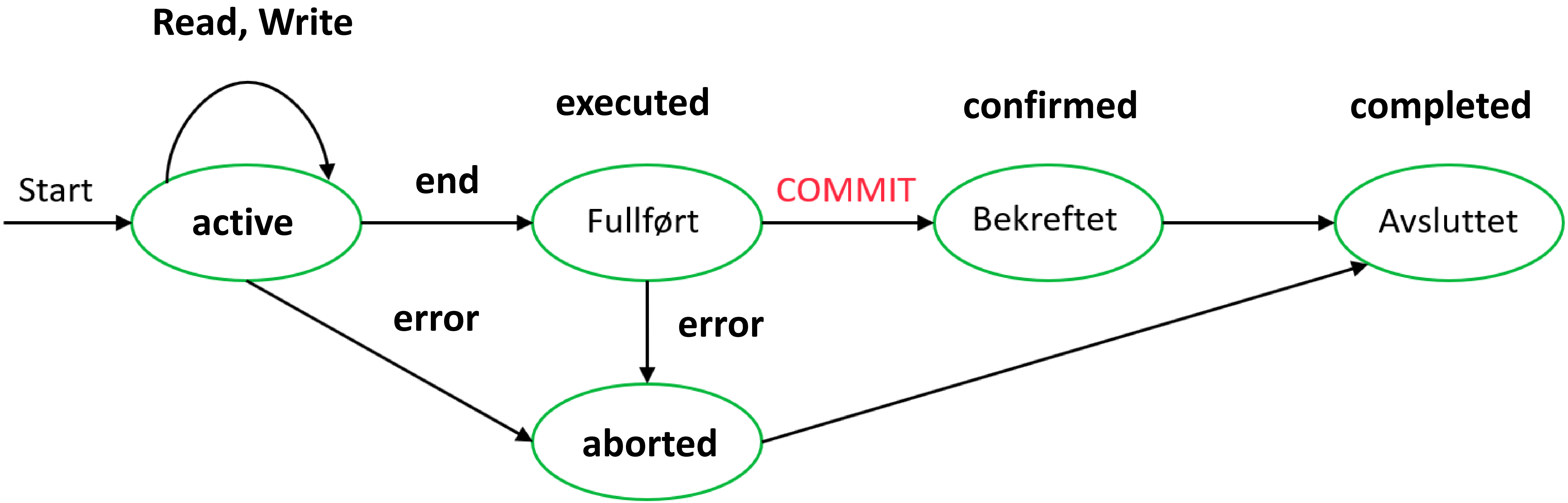
Please answer the practice quizz on mitt.uib now ☺

(you can take it again later if you want)

Link:

- https://mitt.uib.no/courses/27455/quizzes

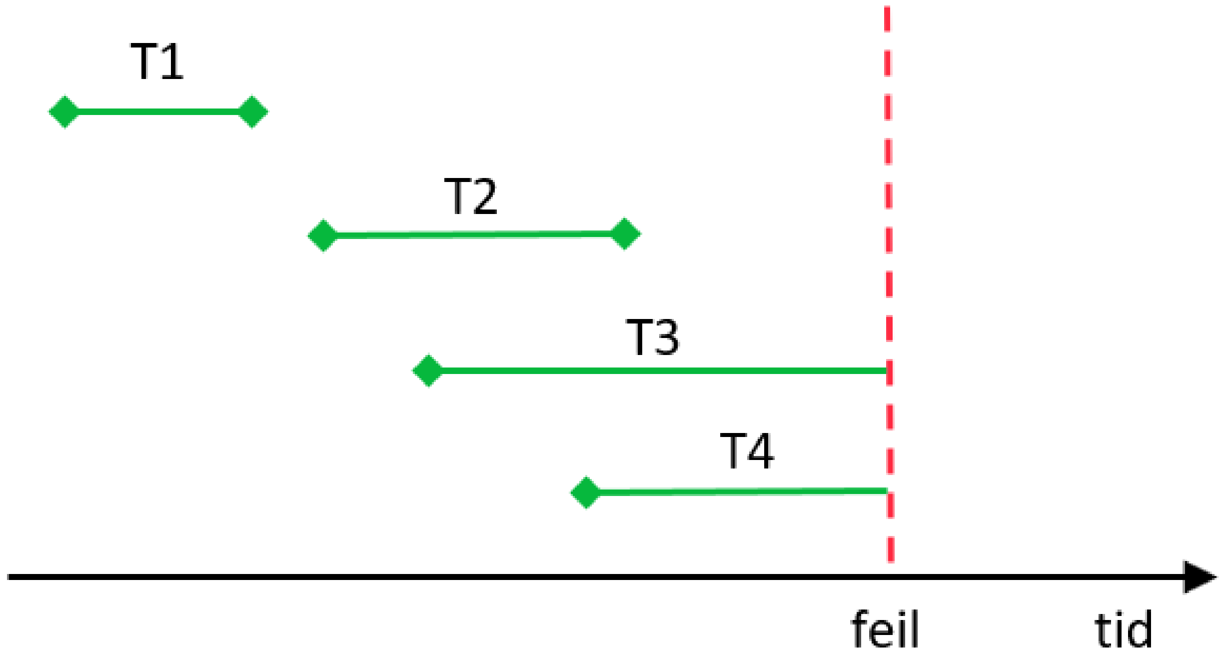
Life cycle of a transaction



Aborting Transactions

Four transactions in a **sequence of events** where an **instance error occurs**.

- ➢ T1 and T2 have written COMMIT to the log and were confirmed before the error occurred.
- ➢ T3 and T4 were running. Perhaps T3 had managed to carry out several writing operations, while T4 had only carried out one reading operation.

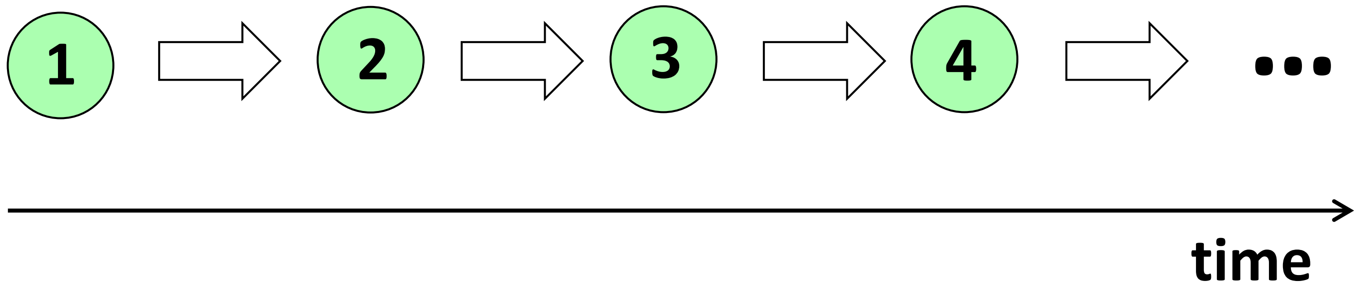


Transactions and States

A **state** is the content of the database at a particular time.

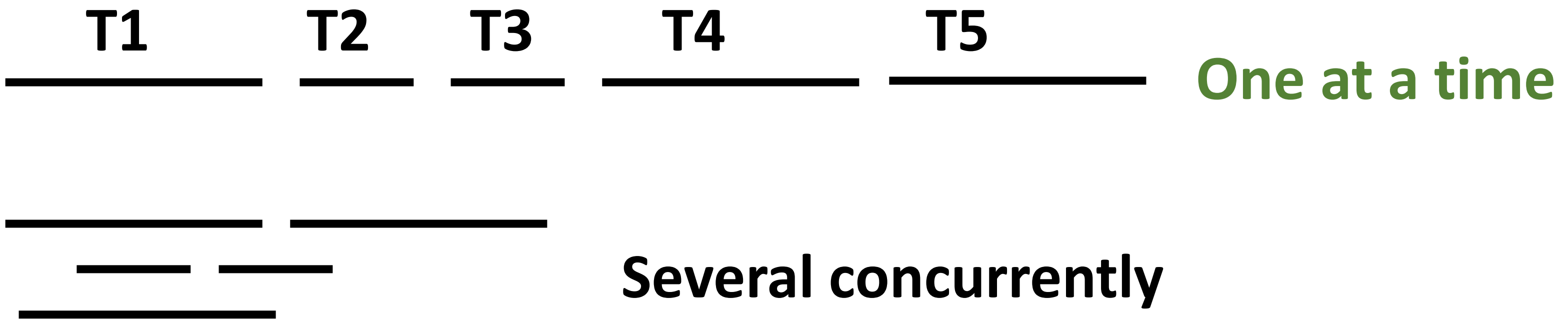
A **transaction** takes the database from one state to another.

If we imagine that *only one transaction is performed at a time*, the **life cycle** of a database can be visualized as:



Concurrency/Simultaneity control

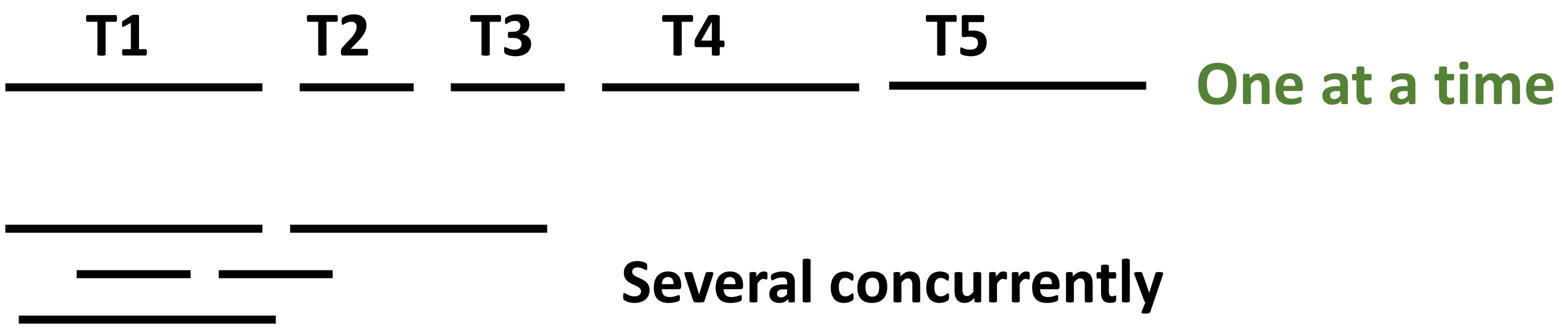
- Want **more simultaneous users / transactions**:
- Utilize **parallelism** (CPU and I / O)
- Reduce average **waiting time** (long transactions)



- **When is it safe** to allow simultaneous access?
- Many users can **read** the same data at the same time.
- Two users can write at the same time only if they work with **different** parts of the database.

Concurrency/Simultaneity control

- ➢ Want **more simultaneous users / transactions**:
- ➢ Utilize **parallelism** (CPU and I / O)
- ➢ Reduce average **waiting time** (long transactions)



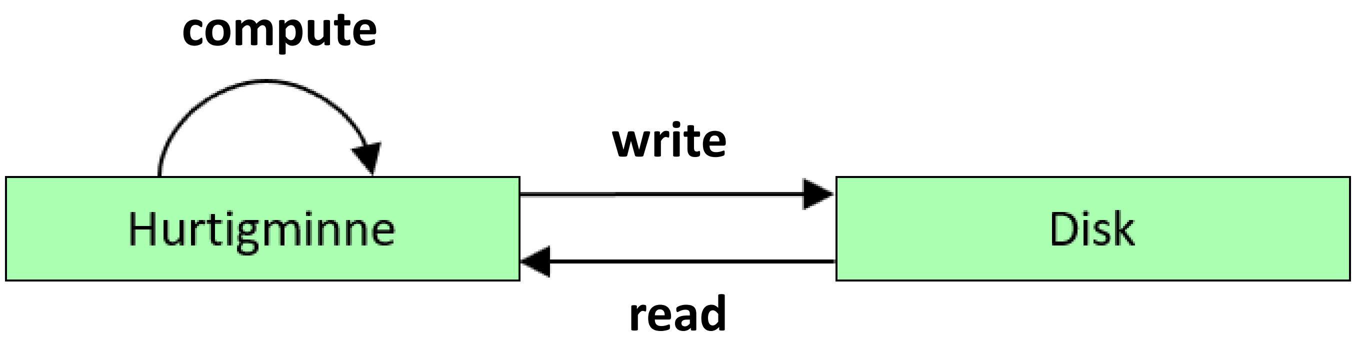
❖ **When is it safe** to allow simultaneous access?

- ➢ Many users can **read** the same data at the same time.
- ➢ Two users can write at the same time only if they work with **different** parts of the database.

Read – Compute – Write

What happens when a «cell» on external storage is to be updated?

```
UPDATE Ansatt
SET Lønn = Lønn * 1.1
WHERE AnsNr = 14
```



Read – Compute – Write

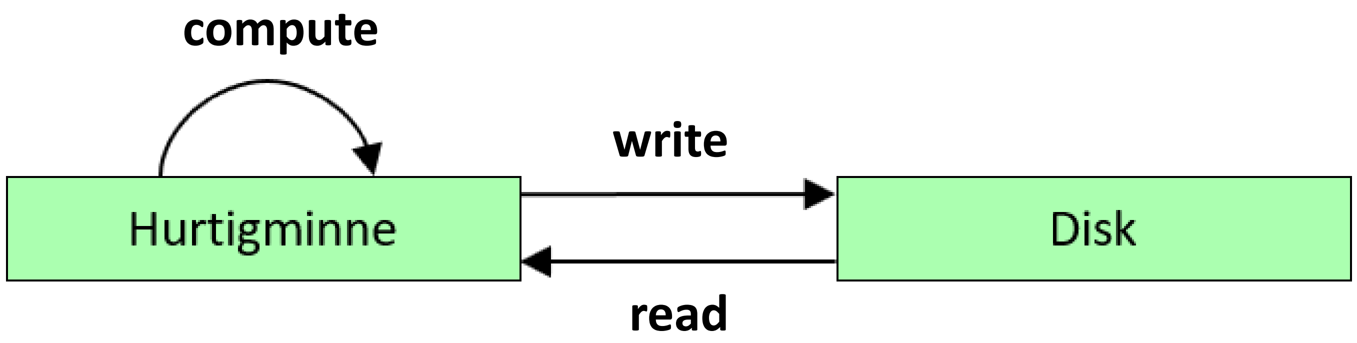
What happens when a «cell» on external storage is to be updated?

```
UPDATE Ansatt
SET Lønn = Lønn * 1.1
WHERE AnsNr = 14
```

The transaction consists of the following sub-operations:

1. **Read** record with AnsNr = 14 from external storage.
2. **Compute** new salary.
3. **Write** result to external storage.

- Read-Compute-Write is **not an atomic operation**.
- Sometimes two updates can be “merged” in time!



Lost Update

- **Simultaneity problems** can occur

  when two transactions work with **the same data at the same time.**

- We study what happens when two transactions are to update a value A on the disk (a value in a specific column in a specific row in a table).

Lokal kopi T1	T1	Verdi A på disk	T2	Lokal kopi T2
120	Les inn	120		
110	Tell ned med 10	120	Les inn	120
110	Skriv til disk	110	Øk med 20	140
		140	Skriv til disk	140

tid

Lost Update

- **Simultaneity problems** can occur

  when two transactions work with **the same data at the same time.**

- We study what happens when two transactions are to update a value A on the disk (a value in a specific column in a specific row in a table).

Lokal kopi T1	T1	Verdi A på disk	T2	Lokal kopi T2
120	Les inn	120		
110	Tell ned med 10	120	Les inn	120
110	Skriv til disk	110	Øk med 20	140
		140	Skriv til disk	140

tid

The **update** to transaction T1 is **overwritten** by transaction T2 and is **lost**.
If A = 120 at **start**, then we get here A = 140 at **end**.
But the **correct result** is A = 130 (120 + 20-10).

Aborted Update (dirty read)

Here, transaction T2 is based on a result from transaction T1 which transaction T1 later "*regrets*" that it produced.

T1	A	T2
	120	
Tell ned A med 10	110	
	110	Les inn A og bruk denne verdien
ROLLBACK	120	

tid

Aborted Update (dirty read)

Here, transaction T2 is based on a result from transaction T1 which transaction T1 later "*regrets*" that it produced.

T1	A	T2
	120	
Tell ned A med 10	110	
	110	Les inn A og bruk denne verdien
ROLLBACK	120	

tid

- **Transactions** must **not be allowed to read other** transactions' **intermediate results**.

Inconsistent analysis (incorrect summary)

Transaction T2 produces a *report* based on some “old” and some “new” results.

T1	A	B	T2	Lokal sum
	10	50	Nullstill sum	0
	10	50	Legg til A i sum	10
Øk A med 10	20	50		10
Øk B med 20	20	70		10
			Legg til B i sum	80

tid

Inconsistent analysis (incorrect summary)

Transaction T2 produces a *report* based on some "old" and some "new" results.

T1	A	B	T2	Lokal sum
	10	50	Nullstill sum	0
	10	50	Legg til A i sum	10
Øk A med 10	20	50		10
Øk B med 20	20	70		10
			Legg til B i sum	80

tid

- Even **one reader** and **one writer** can cause **simultaneity problems**!

15 minute break!

Lecture resumes at 15:10

Locks (Låser)

The general solution to the problems of concurrency:

Transactions must wait for each other.

A **lock** is a «waiting mechanism».

Locks (Låser)

The general solution to the problems of concurrency:

Transactions must wait for each other.

A **lock** is a «waiting mechanism».

A transaction can *lock* larger or smaller parts:

- the entire database
- a table
- a row in a table
- a «cell» in a row in a table

We distinguish between **write locks** (exclusive locks) and **read locks** (shared locks).

Solution: Lost update

Lokal kopi	T1	Verdi A på disk	T2	Lokal kopi
	Skrivelås på A	120		
120	Les inn	120	Skrivelås på A?	
110	Tell ned med 10	120	**Vent**	
110	Skriv til disk	110	**Vent**	
	Lås opp A	110	**Vent**	
		110	Les inn	110
		110	Øk med 20	130
		130	Skriv til disk	130
		130	Lås opp A	

tid

Solution: Lost update

Lokal kopi	T1	Verdi A på disk	T2	Lokal kopi
	Skrivelås på A	120		
120	Les inn	120	Skrivelås på A?	
110	Tell ned med 10	120	**Vent**	
110	Skriv til disk	110	**Vent**	
	Lås opp A	110	**Vent**	
		110	Les inn	110
		110	Øk med 20	130
		130	Skriv til disk	130
		130	Lås opp A	

tid

Aborted updates and *inconsistent analysis* can be solved with locks in respective ways.

Serializability

A **schedule** (forløp) is an arrangement in time (braid) of the sub-operations into a **collection of transactions**.

- Obviously correct: Only 1 transaction at a time = **sequential schedule**.
- **But**: Simultaneous transactions are more efficient.

A **serializable schedule** allows simultaneity («braiding»), but has the same effect as a sequential schedule.

Serializability

A **schedule** (forløp) is an arrangement in time (braid) of the sub-operations into a **collection of transactions**.

- Obviously correct: Only 1 transaction at a time = **sequential schedule**.
- **But**: Simultaneous transactions are more efficient.

A **serializable schedule** allows simultaneity («braiding»), but has the same effect as a sequential schedule.

T1	A	B	T2
Skrivelås A	10	10	Skrivelås B
Tell ned A med 10	0	20	Øk B med 10 hvis større enn 0
Lås opp A	0	20	Lås opp B
Skrivelås B	0	20	Skrivelås A
Tell ned B med 10	0	10	Øk A med 10 hvis større enn 0
Lås opp B	0	10	Lås opp A

tid

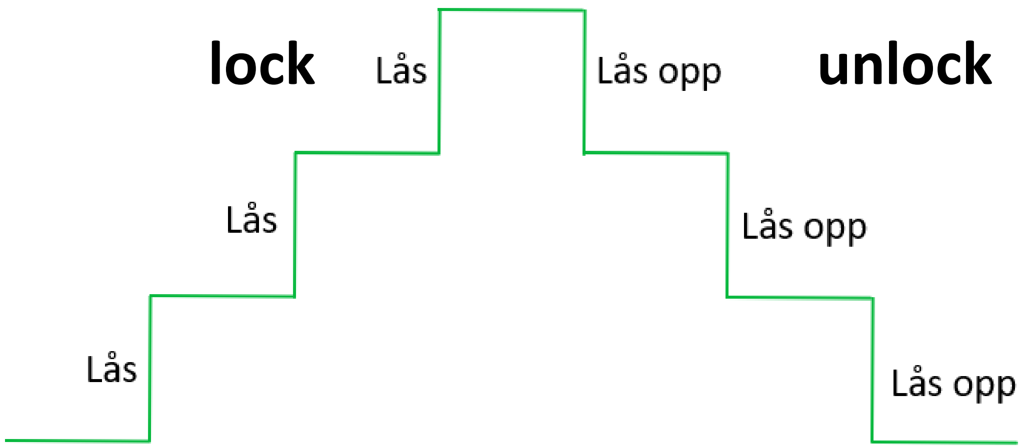
Note: *Not all schedules are serializable even if they use locks !*

Two-Phase Locking

A transaction follows the rules for **two-phase locking**

if all locking operations are performed before the first release (unlocking).

- That is: no locking after the first unlocking!

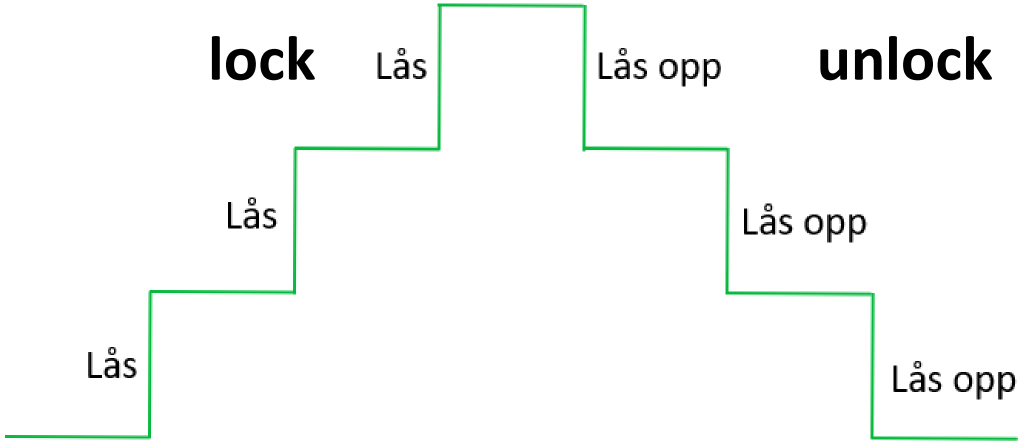


Two-Phase Locking

A transaction follows the rules for **two-phase locking**

if all locking operations are performed before the first release (unlocking).

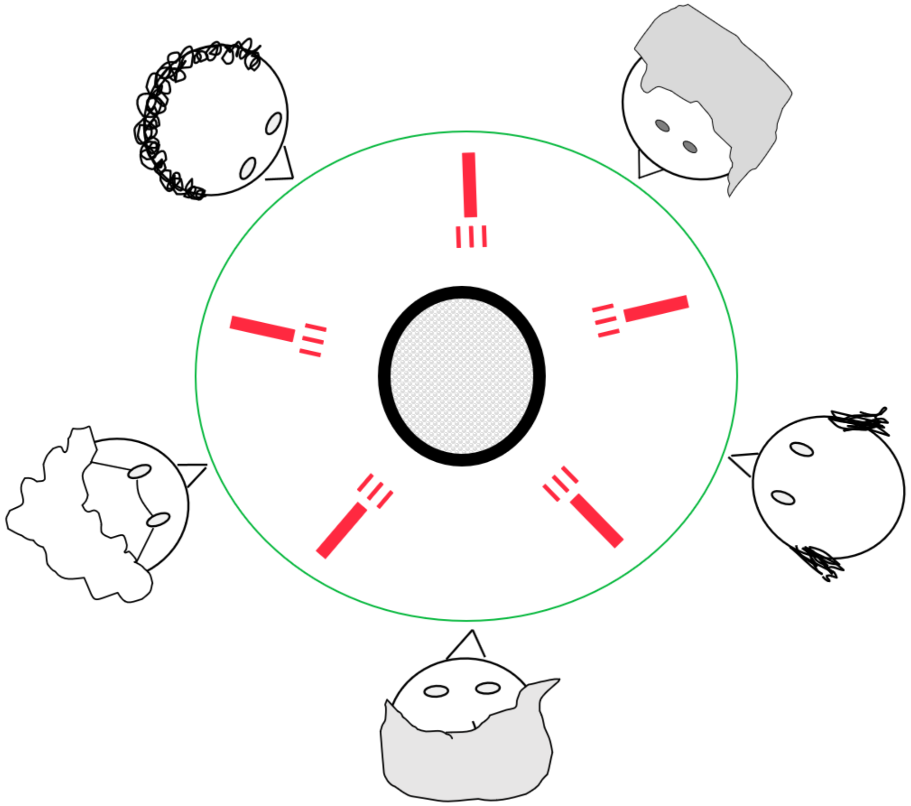
- That is: no locking after the first unlocking!



The following holds: If all transactions follow **two-phase locking**,

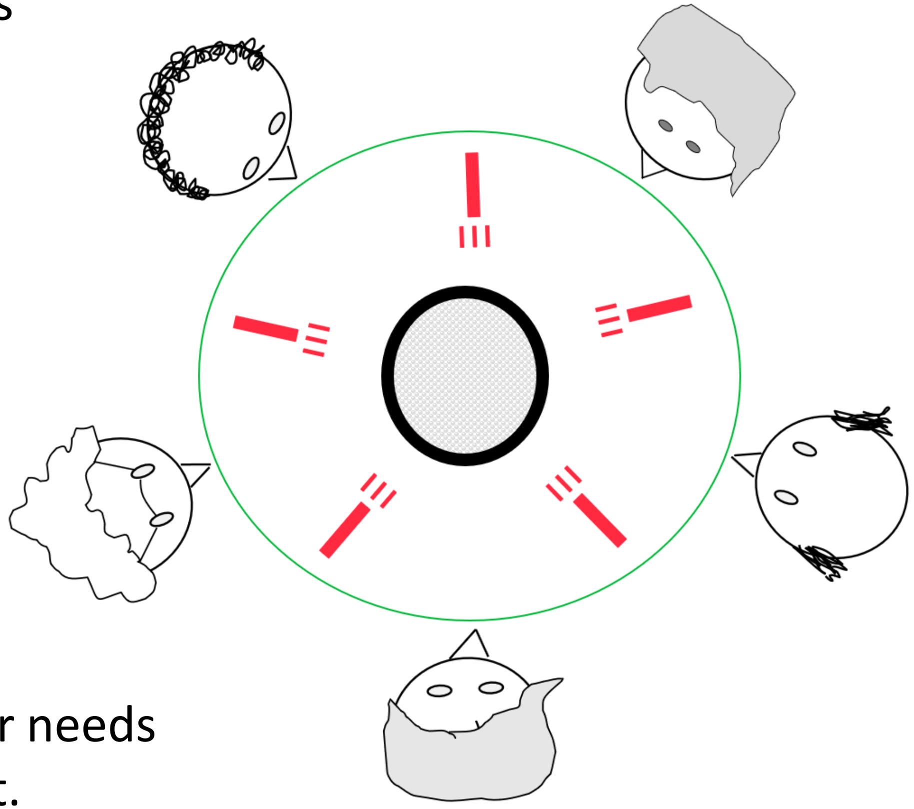
then such a process will be **serializable**.

Deadlock (Vranglås):The Dining Philosophers



Deadlock (Vranglås):The Dining Philosophers

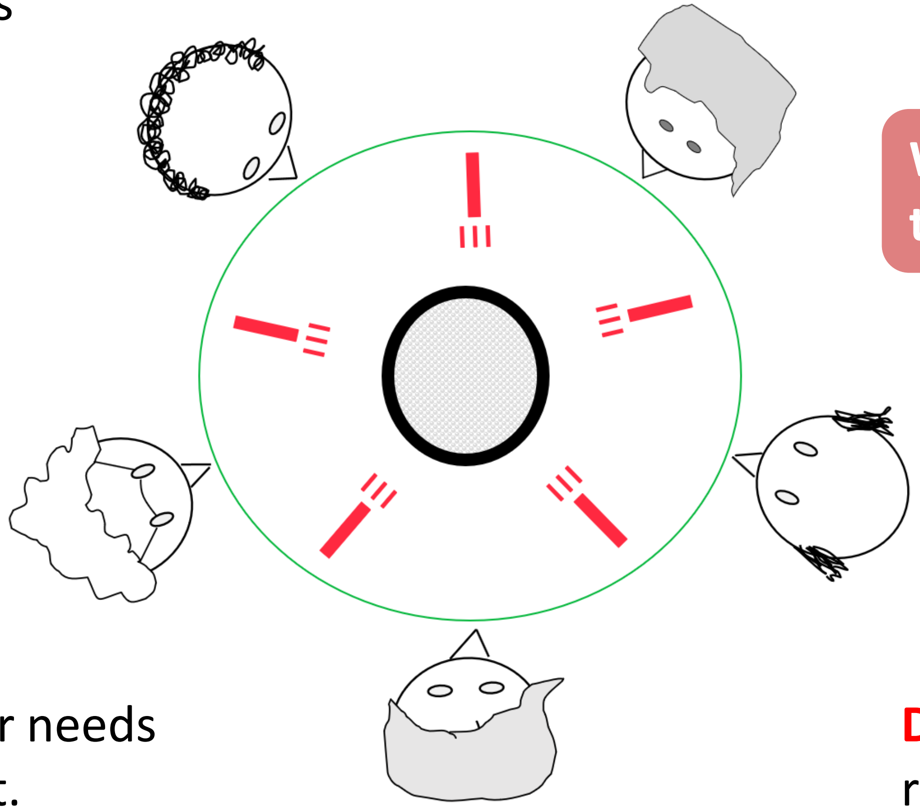
The philosophers
think and **eat**.



Each philosopher needs
both forks to eat.

Deadlock (Vranglås):The Dining Philosophers

The philosophers **think** and **eat**.



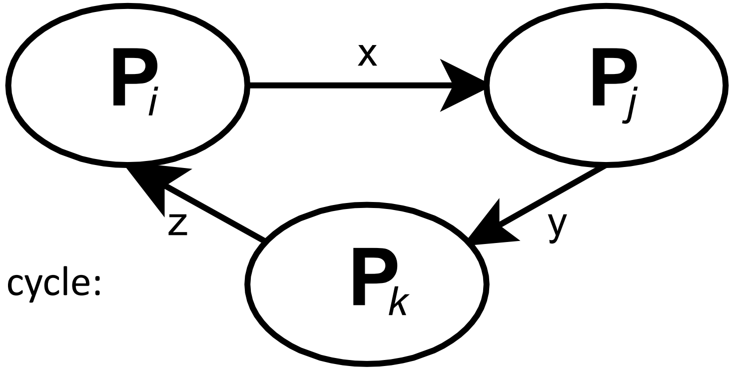
What happens if all want to eat simultaneously ?

Each philosopher needs **both forks** to eat.

Deadlock: Transactions that respectively wait for each other to release the locks.

Managing deadlocks

A simple cycle:



Identifying and resolving deadlocks

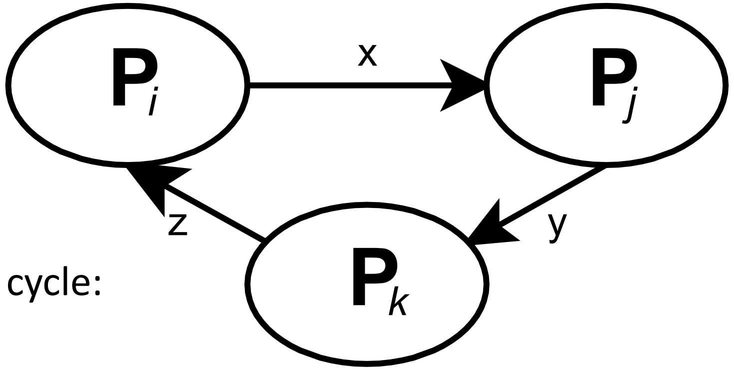
- Build a **wait-for graph**: If transaction P1 has to wait for transaction P2, add an edge (arrow) from P1 to P2.
- Occasionally: Go through the wait-for graph and check if a **cycle** has occurred, for example that T1 is waiting for T2 who is waiting for T3 who is waiting for T1.
- **Select** one of the transactions in the cycle. **Cancel** the transaction ("roll it back"). Complete the others, and start the interrupted transaction afterwards.

Preventing deadlocks

- All transactions are given a unique **timestamp**.
- If a transaction will have to wait for an **older** transaction, it will be **canceled**. This means that younger transactions will never wait for older ones and thus cycles cannot occur.

Managing deadlocks

A simple cycle:



Identifying and resolving deadlocks

- Build a **wait-for graph**: If transaction P1 has to wait for transaction P2, add an edge (arrow) from P1 to P2.
- Occasionally: Go through the wait-for graph and check if a **cycle** has occurred, for example that T1 is waiting for T2 who is waiting for T3 who is waiting for T1.
- **Select** one of the transactions in the cycle. **Cancel** the transaction ("roll it back"). Complete the others, and start the interrupted transaction afterwards.

Preventing deadlocks

- All transactions are given a unique **timestamp**.
- If a transaction will have to wait for an **older** transaction, it will be **canceled**. This means that younger transactions will never wait for older ones and thus cycles cannot occur.

Quizz on *Transactions* (part 2)

Please answer the practice quizz on mitt.uib now ☺

(you can take it again later if you want)

Link:

- https://mitt.uib.no/courses/27455/quizzes

Isolation Levels

Can we accept a little "interference" for efficiency reasons?

- **Dirty Read (Usikker Lesing)**: T1 can read data written by T2 before T2 has written COMMIT.
- **Non-Repeatable Read**: T1 can get two different answers because T2 changes and confirms.
- **Phantoms**: T1 detects new rows inserted by T2.

Isolasjonsnivå	Fantomer	Ikke-repeterbar lesing	Usikker lesing
SERIALIZABLE	Nei	Nei	Nei
REPEATABLE READ	Ja	Nei	Nei
READ COMMITTED	Ja	Ja	Nei
READ UNCOMMITTED	Ja	Ja	Ja

Isolation Levels

Can we accept a little "interference" for efficiency reasons?

- **Dirty Read (Usikker Lesing)**: T1 can read data written by T2 before T2 has written COMMIT.
- **Non-Repeatable Read**: T1 can get two different answers because T2 changes and confirms.
- **Phantoms**: T1 detects new rows inserted by T2.

Isolasjonsnivå	Fantomer	Ikke-repeterbar lesing	Usikker lesing
SERIALIZABLE	Nei	Nei	Nei
REPEATABLE READ	Ja	Nei	Nei
READ COMMITTED	Ja	Ja	Nei
READ UNCOMMITTED	Ja	Ja	Ja

Isolation Levels

Can we accept a little "interference" for efficiency reasons?

- **Dirty Read (Usikker Lesing)**: T1 can read data written by T2 before T2 has written COMMIT.
- **Non-Repeatable Read**: T1 can get two different answers because T2 changes and confirms.
- **Phantoms**: T1 detects new rows inserted by T2.

Isolasjonsnivå	Fantomer	Ikke-repeterbar lesing	Usikker lesing
SERIALIZABLE	Nei	Nei	Nei
REPEATABLE READ	Ja	Nei	Nei
READ COMMITTED	Ja	Ja	Nei
READ UNCOMMITTED	Ja	Ja	Ja

Isolation Levels

Can we accept a little "interference" for efficiency reasons?

- **Dirty Read (Usikker Lesing)**: T1 can read data written by T2 before T2 has written COMMIT.
- **Non-Repeatable Read**: T1 can get two different answers because T2 changes and confirms.
- **Phantoms**: T1 detects new rows inserted by T2.

In SQL: SET TRANSACTION ISOLATION LEVEL

READ COMMITTED;

Isolasjonsnivå	Fantomer	Ikke-repeterbar lesing	Usikker lesing
SERIALIZABLE	Nei	Nei	Nei
REPEATABLE READ	Ja	Nei	Nei
READ COMMITTED	Ja	Ja	Nei
READ UNCOMMITTED	Ja	Ja	Ja

Pessimistic and optimistic locking

- **Pessimistic** locking = standard locking.

- **Optimistic locking:**
  - Appropriate in systems with few conflicts, for example if most people only read, *and* in "interactive" database applications.
  - Transactions are performed without restrictions until COMMIT, but write to local copy.
  - **Validation** before local copy is written to the database.

➢ Can choose between optimistic / pessimistic locking in Access.

Pessimistic and optimistic locking

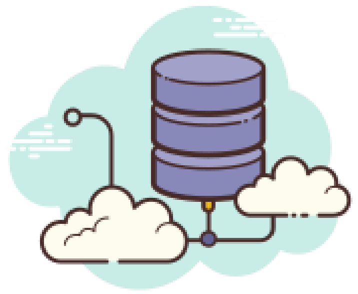
❖ **Pessimistic** locking = standard locking.

❖ **Optimistic locking:**

- Appropriate in systems with few conflicts, for example if most people only read, *and* in "interactive" database applications.
- Transactions are performed without restrictions until COMMIT, but write to local copy.
- **Validation** before local copy is written to the database.

➢ Can choose between optimistic / pessimistic locking in Access.

Detailed Summary: *Transactions*



A **transaction** is a **logical operation**.

- Defined by **COMMIT** and **ROLLBACK**.
- Shall satisfy the **ACID properties**.

➢ *The DBMS must handle **errors** and **concurrency**.*

Errors:

- **Which** transactions were **canceled** when?

➢ **Power outage**: *Transaction log*.

➢ **Disk crash**: *Backup + transaction log*.

Concurrency :

- Transactions must **not** be allowed to ***disrupt*** each other.
- DBMS uses **read** and **write locks**.

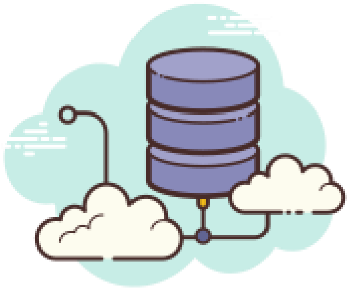
➢ Locks alone do **not** ensure a correct result.

➢ Two-phase locking ensures **serializable processes**.

- **But deadlocks** can happen anyway.

➢ **Deadlocks** can be **detected** or **prevented**.

Detailed Summary: *Transactions*



A **transaction** is a **logical operation**.

- Defined by **COMMIT** and **ROLLBACK**.
- Shall satisfy the **ACID properties**.

➢ *The DBMS must handle errors and concurrency.*

Errors:

- **Which** transactions were **canceled** when?

➢ **Power outage**: *Transaction log*.

➢ **Disk crash**: *Backup + transaction log*.

Concurrency :

- Transactions must **not** be allowed to ***disrupt* each other**.
- DBMS uses **read** and **write locks**.

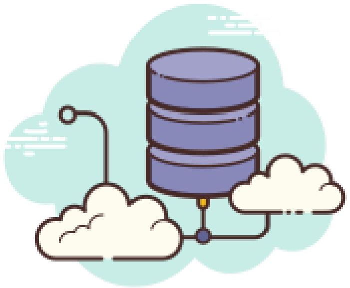
➢ Locks alone do **not** ensure a correct result.

➢ Two-phase locking ensures **serializable processes**.

- **But deadlocks** can happen anyway.

➢ **Deadlocks** can be **detected** or **prevented**.

Detailed Summary: *Transactions*



A **transaction** is a **logical operation**.

- Defined by **COMMIT** and **ROLLBACK**.
- Shall satisfy the **ACID properties**.

➢ *The DBMS must handle errors and concurrency.*

Errors:

- **Which** transactions were **canceled** when?

➢ **Power outage**: *Transaction log*.

➢ **Disk crash**: *Backup + transaction log*.

Concurrency / *Simulataneity*:

- Transactions must **not** be allowed to ***disrupt*** each other.
- DBMS uses **read** and **write locks**.

➢ Locks alone do **not** ensure a correct result.

➢ Two-phase locking ensures **serializable processes**.

- **But** **deadlocks** can happen anyway.

➢ **Deadlocks** can be **detected** or **prevented**.