DBIS Sheet 3

1a.

SHOW transaction_isolation;
-> read commited

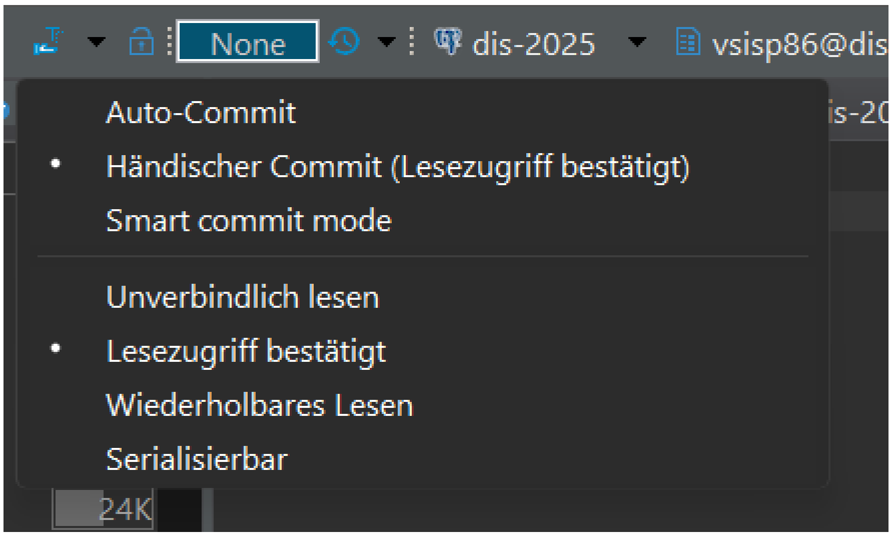
Isolation Level	Description
Read Uncommitted	Not supported – PostgreSQL treats it as **Read Committed**
Read Committed	Default. Sees only data committed before the query started.
Repeatable Read	All reads within the transaction see a consistent snapshot.
Serializable	Strongest isolation. Ensures full serializability using Serializable Snapshot Isolation (SSI).

1b.

```
CREATE TABLE sheet3 (
id SERIAL PRIMARY KEY,
name TEXT
);
```

```
INSERT INTO sheet3 (name) VALUES
('Alice'),
('Bob'),
('Charlie'),
('Daisy');
```

1c.



Auto-Commit
• Händischer Commit (Lesezugriff bestätigt)
Smart commit mode

Unverbindlich lesen
• Lesezugriff bestätigt
Wiederholbares Lesen
Serialisierbar

```
begin;
SELECT * FROM sheet3 WHERE id = 1;
SELECT relation::regclass, mode, granted
FROM pg_locks
WHERE relation::regclass = 'sheet3'::regclass;
```

-> output: sheet3AccessShareLock true
-> currently reading: jemand kann lesen aber niemand kann schreiben bis nicht beendet wurde
-> lightweight lock in PostgreSQL that allows a transaction to read data from a table while preventing schema changes (like dropping or altering the table) but does not block normal data modifications by other transactions.

Lock-Typ	Bedeutung
AccessShareLock	Wird beim Lesen vergeben (SELECT)
RowShareLock	Wird beim Lesen mit `FOR SHARE` gesetzt
RowExclusiveLock	Beim INSERT, UPDATE, DELETE
ShareLock	Wird beim Sperren von Objekten verwendet
ExclusiveLock	Wird z. B. beim ALTER TABLE vergeben

commit;

1.d -> repeat and set to serializable.

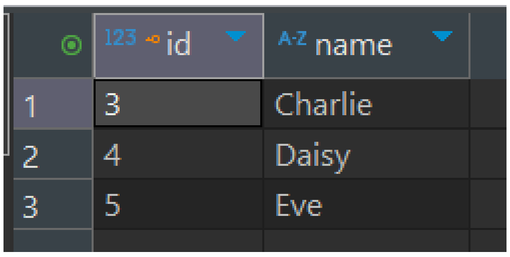
SIreadLock -> is a lock that ensures no other transactions can modify or delete a row while it is being read, providing the highest serializable isolation level for data consistency.

2

2.a.

```
-- In C1------------------------
BEGIN;
SELECT * FROM sheet3 WHERE id > 2;

COMMIT;
-- In C2----------------------
INSERT INTO sheet3 (id, name) VALUES (5, 'Eve');
```

=> 

	id	name
1	3	Charlie
2	4	Daisy
3	5	Eve

c2. Executes and commits immediately adding a new row to the table
In READ COMMITTED isolation level, each SQL statement sees a snapshot of the data as it was when that specific statement began. C2 inserted and committed the row for id=5. When C1 re-ran its SELECT query, this new statement took a new snapshot, which included C2's committed change. This phenomenon, where re-running a query within the same transaction yields different results due to concurrent committed changes, is known as a "non-repeatable read" (specifically, this is a "phantom read" because a new row appeared).

```
-- In C1------------------------
commit;
```

2.b.

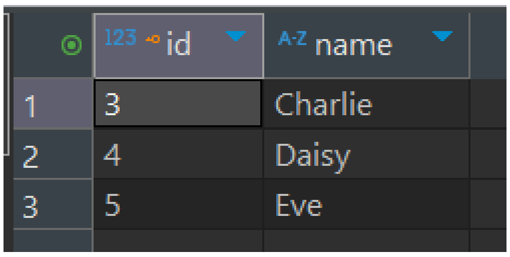
-- In C1-----------------------
BEGIN;
SELECT * FROM sheet3 WHERE id > 2;

-- In C1-----------------------
SELECT relation::regclass, mode, granted, locktype
FROM pg_locks
WHERE relation::regclass = 'sheet3'::regclass AND pid =
pg_backend_pid();

=> sheet3 AccessShareLock true relation
-> PostgreSQL's REPEATABLE READ (and SERIALIZABLE) levels are implemented using Serializable Snapshot Isolation (SSI). For a simple SELECT query, it primarily takes an AccessShareLock on the table, indicating it's being read. It does not necessarily take explicit S-locks on individual rows that would be visible in pg_locks for every row read by a simple SELECT. The repeatability is ensured by the transaction operating on a single snapshot of the data taken at the beginning of the transaction and by tracking read/write dependencies to detect serialization anomalies at commit time.
If PostgreSQL were using a strict lock-based 2PL scheduler for RR, it would conceptually hold shared (S) locks on all rows read (and possibly predicate locks on the query condition id > 2) until the transaction ends.

-- In C2-----------------------
INSERT INTO sheet3 (id, name) VALUES (6, 'TOM');

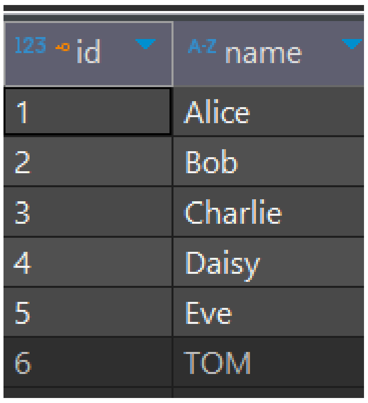
-- In C1-----------------------
SELECT * FROM sheet3 WHERE id > 2;

=> 

	id	name
1	3	Charlie
2	4	Daisy
3	5	Eve

-> In REPEATABLE READ isolation, the transaction operates on a data snapshot taken when the transaction began. All subsequent reads within that transaction see the same data, regardless of concurrent commits by other transactions. This prevents non-repeatable reads and phantom reads.

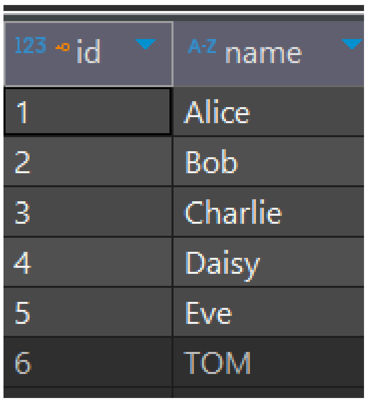
```
-- In C2------------------------
SELECT * FROM sheet3 WHERE id > 2;
```



id	name
1	Alice
2	Bob
3	Charlie
4	Daisy
5	Eve
6	TOM

```
-- In C1 -----------------------
commit;
```

```
-- In C2------------------------
SELECT * FROM sheet3 WHERE id > 2;
```



id	name
1	Alice
2	Bob
3	Charlie
4	Daisy
5	Eve
6	TOM

-> Behaviour is expected

2c.

```
-- In C1 ----------------------
BEGIN;
UPDATE sheet3 SET name = 'Updated by C1' WHERE id = 1;
-- Output: UPDATE 1
```

```
-- In C2 ----------------------
UPDATE sheet3 SET name = 'Updated by C2 (id=2)' WHERE id = 2; -- Auto-commits
```

```
-- In C2 ----------------------
UPDATE sheet3 SET name = 'Updated by C2 (id=1)' WHERE id = 1;
```

-> C2's command will **BLOCK**. It waits because C1 holds an uncommitted exclusive lock on the row with id=1. C2 cannot acquire the necessary lock to update this row until C1 releases its lock (by committing or rolling back).

Does the isolation level matter in this case?

- For this specific scenario of a direct write-write conflict on the same row, the blocking behavior of C2 is fundamental. PostgreSQL will prevent C2 from updating the row that C1 has an uncommitted update on, regardless of whether C1's isolation level is READ COMMITTED, REPEATABLE READ, or SERIALIZABLE. All these levels require exclusive access for updates.
- The isolation level of C1 would be more significant if C1 were performing reads that could be affected by C2, or if C1's commit could fail due to serialization issues in REPEATABLE READ or SERIALIZABLE based on C2's actions (though here C1 locks first).

```
-- In C1 ----------------------
COMMIT;
```

Observation in C2:

- As soon as C1 commits, C2 will unblock. Its UPDATE statement for id=1 will execute and (since auto-commit is ON for C2) commit.

2d.

Beide Connections RC
-- In C1 ----------------------
BEGIN;
UPDATE sheet3 SET name = 'C1 locks id=1' WHERE id = 1;

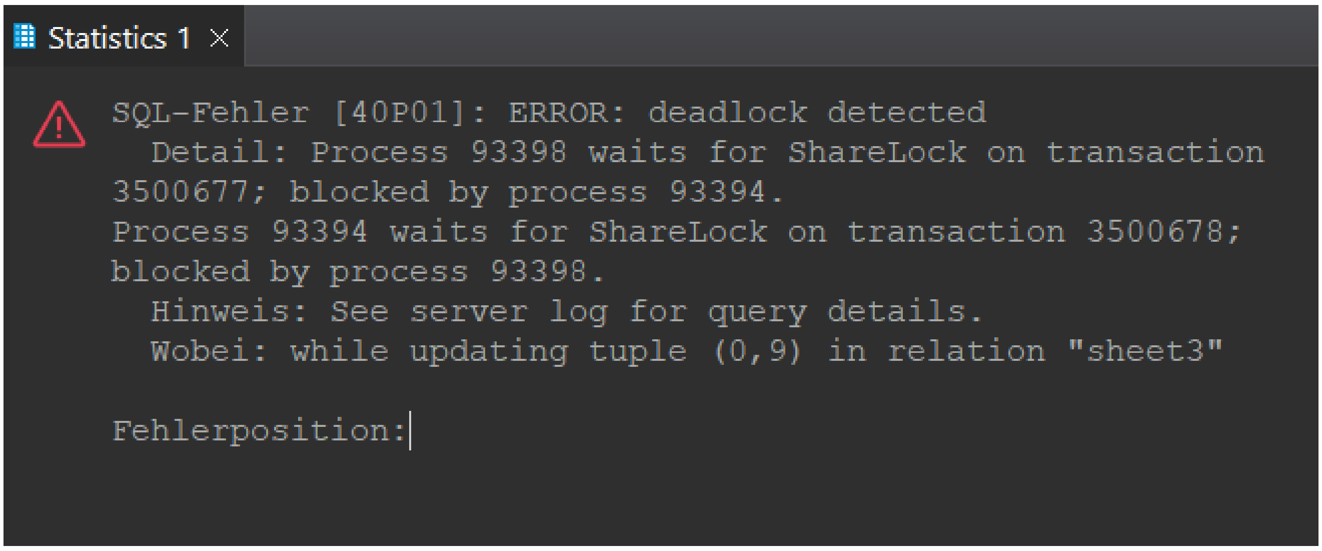
-- In C2 ----------------------
BEGIN;
UPDATE sheet3 SET name = 'C2 locks id=2' WHERE id = 2;

-- In C1 ----------------------
UPDATE sheet3 SET name = 'C1 wants id=2' WHERE id = 2;

Observation: C1 **BLOCKS**, waiting for C2 to release the lock on row id=2.

-- In C2 ----------------------
UPDATE sheet3 SET name = 'C2 wants id=1' WHERE id = 1;

Observation:



```
SQL-Fehler [40P01]: ERROR: deadlock detected
  Detail: Process 93398 waits for ShareLock on transaction
3500677; blocked by process 93394.
Process 93394 waits for ShareLock on transaction 3500678;
blocked by process 93398.
  Hinweis: See server log for query details.
  Wobei: while updating tuple (0,9) in relation "sheet3"

Fehlerposition:
```

Deadlock detected by PostgreSQL and C2 transaction was aborted -> transaction for the victim (one that was aborted) was automatically rollbacked

```
-- In C2 -----------------------
ROLLBACK;
```
(**undo all the changes made by a transaction**)

C1 has unblocked and the update will succeed

```
-- In C1 -----------------------
COMMIT;
```