


| | | | | |
|--|----------------|--|----------|---------------------------|
|  | Course | Databases and Information Systems 2025 | | |
| | Exercise Sheet | 3 | | |
| | Points | – | | |
| | Release Date | April 29 th 2025 | Due Date | May 14 th 2025 |

3 Synchronization with Locking Protocols

Note

- In the following exercises, you can use the psql command-line tool or other tools, e.g., DBeaver or DataGrip. Using these tools makes the administration of two parallel sessions and changing the auto-commit setting more convenient.
- Please briefly note the answers for each question on this sheet and save your used SQL commands.

3.1 Isolation Levels

- Open a connection to database dis-2025. What is the current isolation level? Which isolation levels does PostgreSQL support?
- Create a simple table `sheet3` with columns `id` and `name`. Fill the table with some example data.
- Disable the auto-commit of the current connection, e.g., with the command `\set AUTOCOMMIT off` in psql or via the used UI. Query one row from table `sheet3` and find out the currently held locks. For example, with


```
SELECT relation::regclass, mode, granted
FROM pg_locks
WHERE relation::regclass = 'sheet3'::regclass
```

What locks are held by the transaction? What does this mean?

- Commit the transaction and set the isolation level for the next transaction to `Serializable`. Repeat the steps from c). Which locks are now held by the application?

3.2 Lock Conflicts

- In addition to your existing connection, open a second connection to the database (without disabling the auto-commit). Query some rows via the first connection (isolation level `RC`), for example, all lines with `id > 3` (do **not** complete/commit the transaction of the first connection). Using the second connection, add a new row to the table that satisfies the selection predicate. What happens? Execute the query again via the first connection. What can be observed? Finally, execute a commit.

| | | | | |
|--|----------------|--|----------|---------------------------|
|  | Course | Databases and Information Systems 2025 | | |
| | Exercise Sheet | 3 | | |
| | Points | – | | |
| | Release Date | April 29 th 2025 | Due Date | May 14 th 2025 |

- b) Set the isolation level of the first connecting to RR (still with manual commit). Now repeat the steps from a). What can be observed now? Which locks are held before the commit of the first transaction? What does the table content look like from a new connection before and after the commit of the first transaction has been executed? Is this behavior expected if PostgreSQL would use a lock-based scheduler like 2PL? Explain why or why not. Finally, execute a commit.
- c) Update one row (e.g. id=1) from table sheet3 using the first connection (Still not committing) and change another row's NAME value (e.g. id=2) using the second connection. Then, additionally, change the same row in transaction 2 as in transaction 1 (e.g. id=1). What happens? Does the isolation level matter in this case? Finally, execute a commit.
- d) Using two connections with isolation levels of your choice: Which actions lead to an abort/rollback? Describe the situation in which the database needs to abort the transaction and give an example.

3.3 Scheduling


- a) Using a language of your choice (such as Python or Java), create a script that executes the given schedules using two connections per schedule (one connection per transaction). In Moodle, you will find an example project where S1 is already implemented in SQL. You can use this or create your application. The application shall simulate two processes that access the same database. The provided schedules define the order in which the commands reach the database. Using RC as isolation mode, describe briefly what happens and which values are in the table after the execution of each schedule.

```

S1 = r1(x) w2(x) c2 w1(x) r1(x) c1
S2 = r1(x) w2(x) c2 r1(x) c1
S3 = r2(x) w1(x) w1(y) c1 r2(y) w2(x) w2(y) c2

```

- b) Set both transactions in your software to Serializable. **Does the result differ? If yes, why?** Describe what you observed.
- c) Now, use Read Committed again as the isolation level. Adapt the program/script to handle the lock management row-wise (RX Locking). Normally, the database system handles the scheduling; for this task, your application shall ensure a serializable schedule by setting locks. **Achieve serializability by using SS2PL.** For this, let your program open two connections (RC) and set the corresponding locks for operations.

| | | | | |
|--|----------------|--|----------|---------------------------|
|  | Course | Databases and Information Systems 2025 | | |
| | Exercise Sheet | 3 | | |
| | Points | – | | |
| | Release Date | April 29 th 2025 | Due Date | May 14 th 2025 |

You can read-lock(shared lock) one row with `SELECT * FROM sheet3 WHERE id = 3 FOR SHARE;` or update-lock (exclusive lock) with `SELECT * FROM sheet3 WHERE id = 3 FOR UPDATE;`.

Each lock command is an independent command. **You still need to run the READS/WRITES after acquiring the locks.** This means, to lock and read or lock and write, you need two commands (the lock with select, from which you can ignore the returned value, and the command itself.)

Example for read:

```
SELECT * FROM sheet3 WHERE id = 3 FOR SHARE;
SELECT name FROM sheet3 WHERE id = 3;
```

Example for write:

```
SELECT * FROM sheet3 WHERE id = 3 FOR UPDATE;
UPDATE sheet3 SET name = 'Daisy' WHERE id = 3;
```

PostgreSQL does unlock all rows at the end of a transaction.

Execute the same schedules, do not alter the execution order manually. Print out the executed serial schedule.

Does the result differ from the serialized result from PostgreSQL? If yes, why?

- d) **Optional:** Implement 2PL; you can't directly use the same locks as before. What's the issue? What are the alternatives?