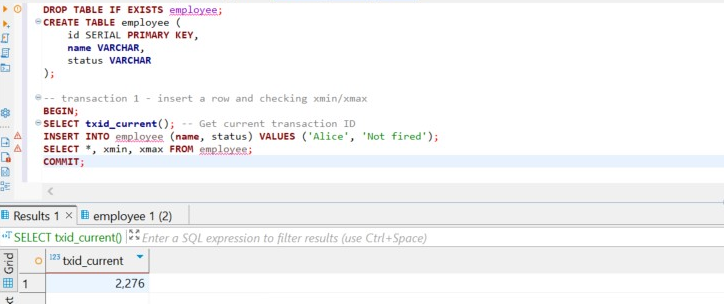
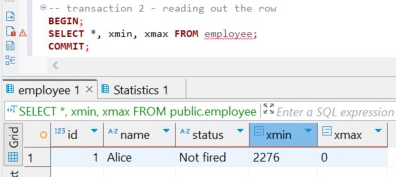
# 1. Create a table called employee with columns id serial, name varchar, status varchar.



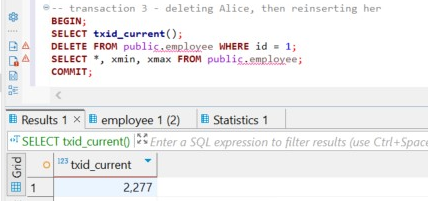
2. Replicate the example given in the lecture with the code below:

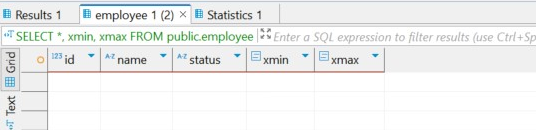
## 1)First transaction



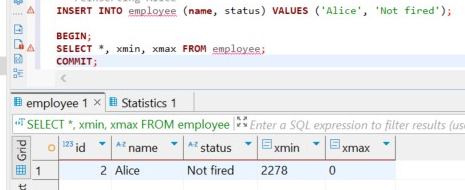
We get that xmin = 2276, which is the transaction ID that created the row. xmax = 0: because it's still alive.

## 2) Second transaction



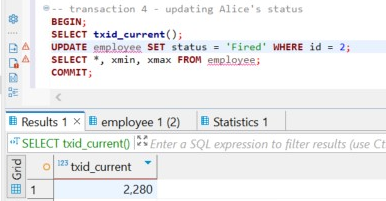


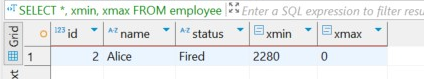
Since we deleted the row, it's invisible to future queries. Its xmax will be 2278. (forgot to delete the schema name in front of the table).

After reinserting the row, we get  


That means that we took the new space where the xmin = 2278 and xmax = 0, since it’s still alive.

## 3) third transaction





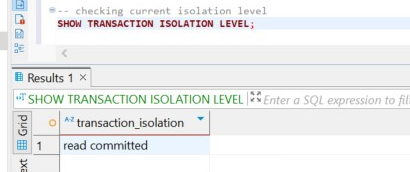
I accidentally ran this query twice, so its our current transaction number is 2280. After updating the row, it gets xmin value of 2280 and xmax = 0, since its alive. Transactions 2278 and 2279 are dead.

# 3. Run the command set transaction isolation level REPEATABLE READ.

**SET** **TRANSACTION** **ISOLATION** **LEVEL** **REPEATABLE** **READ**;

After doing this, all reads within the transaction will see the snapshot taken at BEGIN. Even if others commit changes, we will not be able to see them.

# **4. Check your current isolation level in each session with show transaction isolation level.**



We get that our current isolation level is read commited (because we ran the previous quey)

# 5. Recreate employee table and redo the second task but modify the code so that select statements would now include cmin and cmax system columns. What changed?

**DROP** **TABLE** **IF** **EXISTS** employee;

**CREATE** **TABLE** employee(

id SERIAL **PRIMARY** **KEY**,

name **VARCHAR**,

status **VARCHAR**

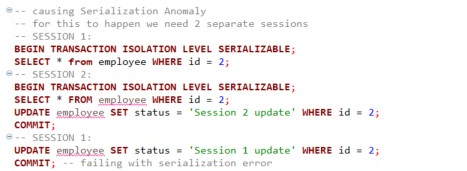
);

Here we ecreated the employee table and repeated the previous transaction operations, but now we attempt to include two additional system columns: cmin and cmax. These columns are internal to PostgreSQL and serve a different purpose than the previously used xmin and xmax.

While xmin and xmax refer to the transaction IDs that inserted or deleted a row, cmin and cmax represent the command identifiers within a single transaction. Specifically, cmin indicates the command within the transaction that inserted the row, and cmax refers to the command that deleted it. Attempting to select these columns directly (e.g., SELECT cmin, cmax FROM employee) results in an error because they are not exposed in the user-facing schema. To access them we need extension *pageinspect.*

# 6) Try to cause a serialization anomaly on the employee table (add more data if necessary). Change your isolation level to serializable and try to cause serialization anomaly one more time. What happened?

after dropping and recreating the employee table , for causing the serialization error, we have to run two separate sessions, for this to happen I turned off auto commit and ran first session without commiting, after it I ran the second session and commited it. After all of this I returned to first session and commited it. This was the way I caused the serialization error.



A **serialization error** occurs when a transaction running under the SERIALIZABLE isolation level **cannot safely commit** because another transaction has modified the same data in a way that **would lead to inconsistent or incorrect results.**  SERIALIZABLE mode in PostgreSQL uses Serializable Snapshot Isolation (SSI). It tracks **read/write dependencies** between transactions to ensure that no conflicts occur. If a conflict is detected, such as read–write or write–write, PostgreSQL raises a serialization error rather than allowing the conflict to cause **data anomalies**.

To cut it short when two transactions read the same data and try to update it, one is aborted to prevent inconsistency. PostgreSQL protects data integrity by issuing a **serialization error**, ensuring the system behaves as if transactions were run sequenti

# **7) \* Set your isolation level to read committed. Try to cause a lost update database anomaly on the employee table (add more data if necessary). What happened? What do you think are the downsides of the approach that Postgres took to handle this anomaly**

# 

While using default isolation level, READ COMMITED, each SELECT sees the latest committed data **at the time the query runs**, but there is **no lock or check** between the time of read and write, and because of that the update in Session A goes through, even though it was based on stale data. That is exactly what happened in our case. To prevent session 2 from silently overwriting the session 1, we can write

SELECT status FROM employee WHERE id = 2 FOR UPDATE;

This locks row until SESSION 1 commits

Now, if Session 2 tries to update the same row, it will wait or fail, depending on lock timeout.

In conclusion PostgreSQL’s default isolation level, does **not prevent lost updates**. When two sessions read and then update the same row without locking, one update silently overwrites the other. PostgreSQL allows this for performance reasons but places the burden of conflict prevention on the developer (e.g., using SELECT FOR UPDATE).