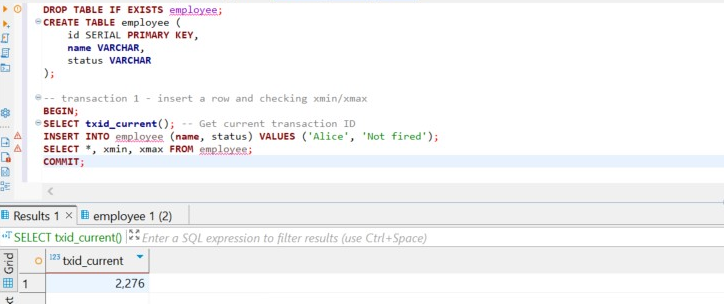
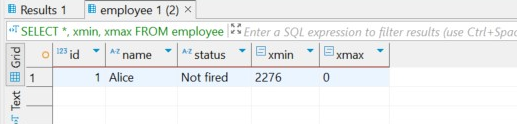
Replicating the queries

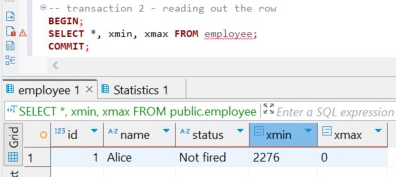
1) transaction 1





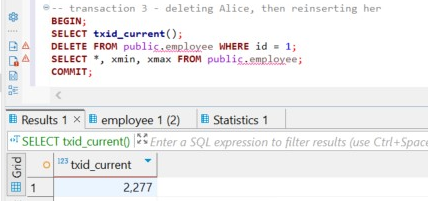
We get that xmin = 2276, which is the transaction ID that created the row. xmax = 0: because it's still alive.

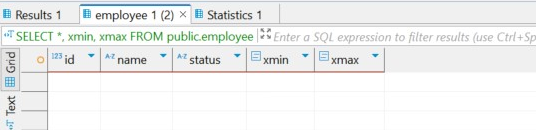
2) transaction 2



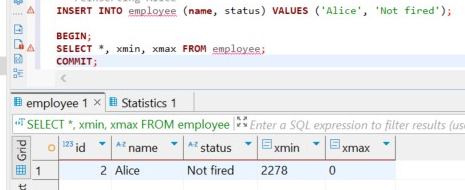
Gives us the same result as the second part of the first transaction.

3) transaction 3



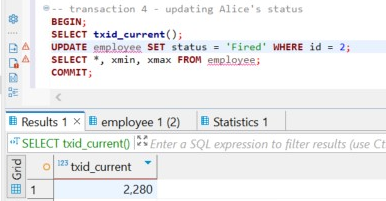


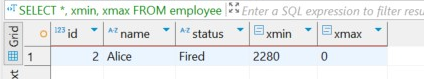
Since we deleted the row, it's invisible to future queries. Its xmax will be 2278. (forgot to delete the schema name in front of the table).

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.

4) transaction 4’



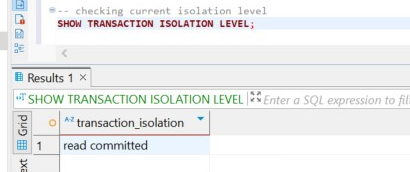


I accidentally ran this query twice, so its our current transaction number is 2280. After updating the row, it gets xmin value of 2280 and xmax = 0, since its alive. Transactions 2278 and 2279 are dead.

5) Setting the transaction isolation level to REPEATABLE READ

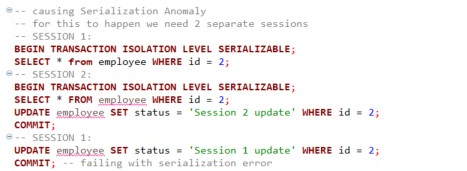
After doing this, all reads within the transaction will see the snapshot taken at BEGIN. Even if others commit changes, we will not be able to see them

6) checking the current isolation level



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

7) after dropping and recreating the employee table and recreating it, 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**.

8) lost update



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.