1 Stored Procedures und Trigger

Implementieren Sie die folgende Funktion als **Stored Procedure** auf dem Datenbankserver und testen Sie diese. Verwenden Sie hierzu PL/pgsql.

- Eine Funktion isFreeSlot, welche einen Fernsehsender und ein Zeitintervall übergeben bekommt und berechnet, ob innerhalb dieses Intervalls eine weitere Ausstrahlung erfolgen kann oder sich diese mit bereits angesetzten Ausstrahlungen überlappen würde.

Implementieren Sie weiterhin einen **Trigger**, welcher neue Ausstrahlungen nur dann zulässt, wenn die entsprechenden Slots noch frei sind. Ist der Slot nicht komplett frei, so soll das entsprechende Insert **abgewiesen** werden.

Testen Sie Ihren Trigger mit sinnvollen Beispieldaten. Achten Sie darauf, dass Sie jederzeit mit Ihren erzeugten Skripten auf den Ausgangszustand zurücksetzen können.

1. Erstellen der Funktion isFreeSlot

```
DROP FUNCTION isfreeslot(character varying,timestamp without time
zone,timestamp without time zone);

CREATE OR REPLACE FUNCTION isFreeSlot(
    function_name VARCHAR,
    function_startzeit TIMESTAMP,
    function_endzeit TIMESTAMP
) RETURNS BOOLEAN AS $isFreeSlot$
DECLARE
    overlapping_count INTEGER;
BEGIN
    SELECT COUNT(*)
    INTO overlapping_count
    FROM ausstrahlung
    WHERE name = function_name
      AND function_startzeit < endzeit
      AND function_endzeit > startzeit;

    IF overlapping_count > 0 THEN
        RETURN FALSE;
    ELSE
        RETURN TRUE;
    END IF;
END;
$isFreeSlot$ LANGUAGE plpgsql;
```

2. Erstellen der Trigger-Funktion

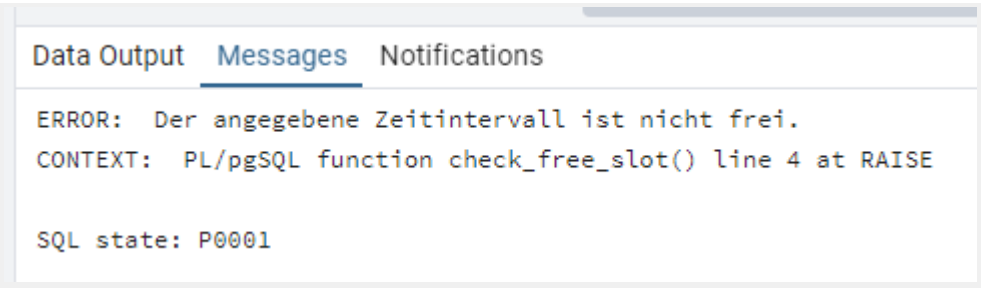
```
CREATE OR REPLACE FUNCTION check_free_slot() RETURNS TRIGGER AS $$
BEGIN
    IF NOT isFreeSlot(NEW.name, NEW.startzeit, NEW.endzeit) THEN
        RAISE EXCEPTION 'Der angegebene Zeitintervall ist nicht frei.';
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

3. Erstellen des Triggers

```
CREATE TRIGGER trg_check_free_slot
BEFORE INSERT ON ausstrahlung
FOR EACH ROW
EXECUTE FUNCTION check_free_slot();
```

4. Testen:

```
INSERT INTO ausstrahlung (serienname, nummer, staffel, name,
ausstrahlungsnummer, startzeit, endzeit) VALUES ('Breaking Bad', 1, 1,
'SAT1', 1, '2023-05-01 15:30:00', '2023-05-01 16:30:00');
```



2 Views

Return to ~~Monke~~ Excel: Verfassen Sie eine View, welches Ihre gesamte Datenbank denormalisiert in eine Tabelle “platt klopft”, damit Sie diese herunterladen und Ihrem Abteilungskollegen ohne DB-Hintergrund als Excel-Datei schicken können ;)

```
set search_path to streaming;
CREATE OR REPLACE VIEW denormalized_view AS
SELECT
k.id as Kunstler_id,
k.vorname as Kunstler_vorname,
k.nachname as Kunstler_nachname,

hh.serienname as hauptschauspieler_in_serie,
hh.id as hauptschauspieler_id,

s.serienname as stoffentwickler_in_serie,
s.id as stoffentwickler_id,

e.serienname as episode_serie,
e.staffel as episode_staffel,
e.nummer as episode_nr,
e.titel as episode_titel,
e.imdbrating as episode_rating,

se.name as sender_name,

a.serienname as ausstrahlung_serie,
a.nummer as ausstrahlung_serie_nummmer,
a.staffel as ausstrahlung_serie_staffel,
a.name as ausstrahlung_sender,
a.ausstrahlungsnummer as ausstrahlung_nummer,
a.startzeit as ausstrahlung_startzeit,
a.endzeit as ausstrahlung_endzeit,

ba.name as bietetan_streamingdiemst,
ba.serienname as bietetan_serie,
ba.staffel as bietetan_serie_staffel,
ba.nummer as bietetan_serie_nummer,

sd.name as streamingdienst,

v.name as vertrag_streamingdiemst,
v.email as vertrag_kunde_email,
v.monatspreis as vertrag_monatspreis,
v.vertragslaufzeit as vertrag_laufzeit,

ku.typ as kunde_typ,
ku.email as kunde_email
```

```
FROM
    episode e
    FULL OUTER JOIN serie s ON e.serienname = s.serienname
    FULL OUTER JOIN hat_hauptschauspieler hh ON s.serienname = hh.serienname
    FULL OUTER JOIN kunstler k ON hh.id = k.id
    FULL OUTER JOIN ausstrahlung a ON e.serienname = a.serienname AND e.nummer =
a.nummer AND e.staffel = a.staffel
    FULL OUTER JOIN sender se ON a.name = se.name
    FULL OUTER JOIN bietet_an ba ON e.serienname = ba.serienname AND e.nummer =
ba.nummer AND e.staffel = ba.staffel
    FULL OUTER JOIN vertrag v ON ba.name = v.name
    FULL OUTER JOIN streamingdienst sd ON v.name = sd.name
    FULL OUTER JOIN kunde ku ON v.email = ku.email;

-- Daten der aktualisierten View anzeigen
SELECT * FROM denormalized_view;
```

3 Transaktionsmanagement – Isolationlevel im konkurrierenden Zugriff zweier Sessions

1. Setzen Sie in pgAdmin "autocommit" auf "false":

   → File / Preferences / (Query Tool) / Options / hier die 1. Option: Auto Commit=False

2. Starten Sie zu Beginn **zwei PostgreSQL-Sessions** (→ QueryTool, im selben Browser oder auch auf unterschiedlichen Rechnern).

3. Prüfen Sie in jeder Session, ob Sie auch tatsächlich unterschiedliche Session IDs haben:
   select pg_backend_pid();

   Beenden Sie dann die aktuelle Transaktion mit
   commit();

   Führen Sie in zwei unterschiedlichen „Experimenten" die unter 4. genannten Operationen zweier konkurrierender Transaktionen aus, jeweils unter einem gemeinsamen Isolationlevel: Zunächst unter dem Isolationlevel SERIALIZABLE, anschließend unter dem Isolationlevel READ COMMITTED. Verwenden Sie jeweils die selbe Einstellung für beide Sessions!

Der Isolationlevel wird jeweils zu Beginn einer Session für jede Transaktion wie folgt definiert:

BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

bzw.

BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

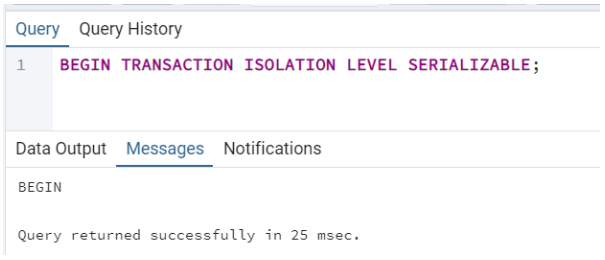
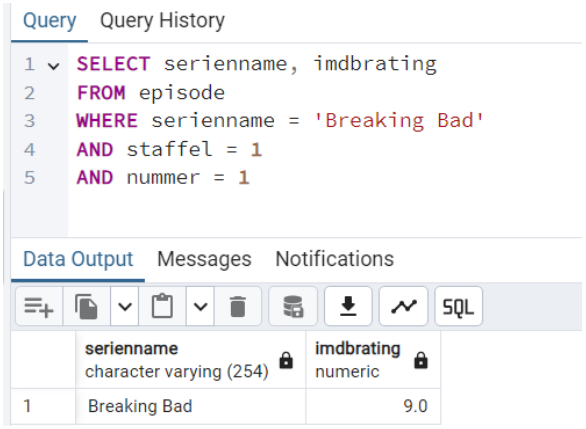
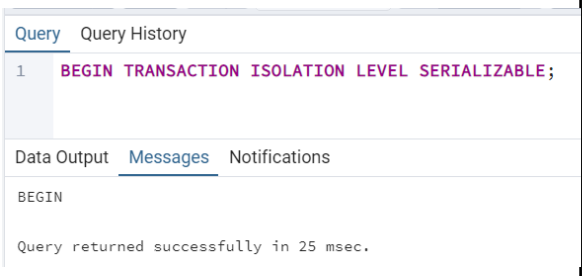
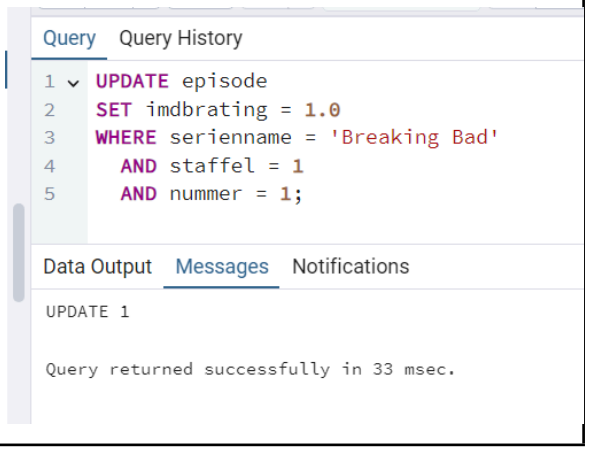
4. Führen Sie die Operationen jeweils auf der gleichen Tabelle, z.B. für eine bestimmte Station, innerhalb der beiden offenen Sessions auf Ihrer Datenbank „Serien-Streaming" entsprechend der nachfolgenden Tabelle interaktiv auf einem **eindeutigen** record aus:

	SQL-Developer I	SQL-Developer II
1	**BOT T1**	
2	T1 liest record	
3		**BOT T2**
4		T2 schreibt record
5		**Commit T2**
6	T1 liest record	
7	T1 schreibt record	
8	T1 liest record	
9	**Commit T1**	

Protokollieren Sie genau die Ergebnisse der beiden Transaktionen auf der Datenbank unter den beiden Isolationleveln **Serializable** und **Read Committed**.

Welches Verhalten bzw. Ergebnis pro Operation erwarten Sie jeweils? Erklären Sie das beobachtete, pro Isolationlevel unterschiedliche Verhalten – entspricht es Ihren Erwartungen?

6. Führen Sie **zwei konkurrierende Transaktionen** in zwei parallelen Sessions (pgAdmin) auf Ihrer Datenbank so aus, dass ein **Dead Lock** entsteht.

	SQL Developer 1	SQL Developer 2
1		
2		
3		
4		

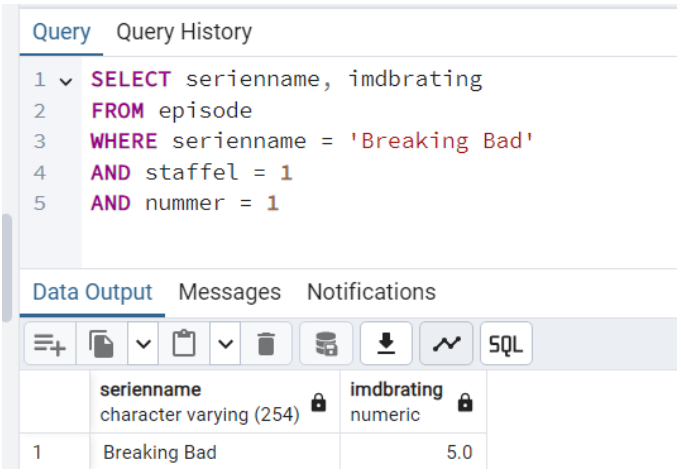
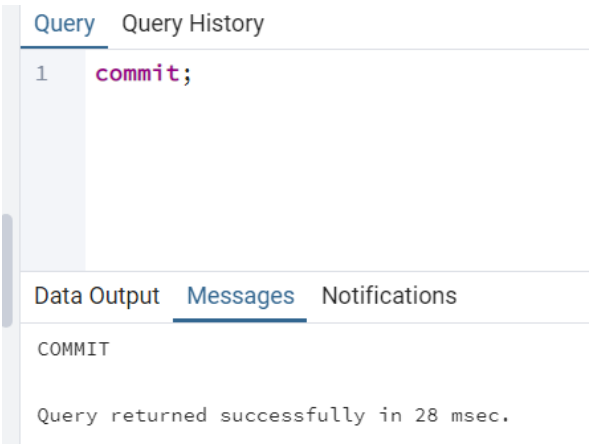
5		Query Query History 1 commit; Data Output Messages Notifications COMMIT Query returned successfully in 68 msec.
6	Query Query History 1 SELECT serienname, imdbrating 2 FROM episode 3 WHERE serienname = 'Breaking Bad' 4 AND staffel = 1 5 AND nummer = 1 Data Output Messages Notifications SQL serienname character varying (254) 🔒 imdbrating numeric 🔒 1 Breaking Bad 9.0	
7	Query Query History 1 UPDATE episode 2 SET imdbrating = 1.0 3 WHERE serienname = 'Breaking Bad' 4 AND staffel = 1 5 AND nummer = 1; Data Output Messages Notifications ERROR: could not serialize access due to concurrent update SQL state: 40001	
8	Query Query History 1 SELECT serienname, imdbrating 2 FROM episode 3 WHERE serienname = 'Breaking Bad' 4 AND staffel = 1 5 AND nummer = 1 Data Output Messages Notifications ERROR: current transaction is aborted, commands ignored until end of transaction block SQL state: 25P02	

9	Query Query History 1 commit; Data Output Messages Notifications ROLLBACK Query returned successfully in 85 msec.	

READ COMMITTED

	SQL Developer 1	SQL Developer 2
1	Query Query History 1 BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED; Data Output Messages Notifications BEGIN Query returned successfully in 45 msec.	
2	Query Query History 1 SELECT serienname, imdbrating 2 FROM episode 3 WHERE serienname = 'Breaking Bad' 4 AND staffel = 1 5 AND nummer = 1 Data Output Messages Notifications SQL serienname character varying (254) / imdbrating numeric 1 Breaking Bad 9.0	
3		Query Query History 1 BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED; Data Output Messages Notifications BEGIN Query returned successfully in 45 msec.

4		Query Query History 1 UPDATE episode 2 SET imdbrating = 1.0 3 WHERE serienname = 'Breaking Bad' 4 AND staffel = 1 5 AND nummer = 1; Data Output Messages Notifications UPDATE 1 Query returned successfully in 71 msec.
5		Query Query History 1 commit; Data Output Messages Notifications COMMIT Query returned successfully in 25 msec.
6	Query Query History 1 SELECT serienname, imdbrating 2 FROM episode 3 WHERE serienname = 'Breaking Bad' 4 AND staffel = 1 5 AND nummer = 1 Data Output Messages Notifications SQL serienname character varying (254) / imdbrating numeric 1 Breaking Bad 1.0	
7	Query Query History 1 UPDATE episode 2 SET imdbrating = 5.0 3 WHERE serienname = 'Breaking Bad' 4 AND staffel = 1 5 AND nummer = 1; Data Output Messages Notifications UPDATE 1 Query returned successfully in 40 msec.	

8		
9		

SERIALIZABLE:

- Nach dem T1 Read wurde T2 gestartet welche einen WRITE gemacht hat
- Nach dem WRITE und Commit von T2, war die Ressource überschrieben.
- In T1 wurde mit einem zweiten kein neuer Wert gelesen => non-repeatable READ verhindert, es wurde der Wert vom Anfang von T1 ausgegeben
- Beim Versuch zu schreiben von T!, erkennt das System einen Konflikt, da das Tupel inzwischen von T2 geändert wurde.
- Nach fehlgeschlagenem WRITE von T1 konnte kein READ von T1 mehr ausgeführt werden, die Transaction wurde abgebrochen und es wurde eine ERROR nachricht ausgegeben, welche mitteilte, dass Commands bis zum Ende der Transaktion ignoriert werden
- Nach COMMIT von T1 wurde ein ROLLBACK gemacht.
- (Keine Transaktion gleichzeitig, nacheinander)

READ COMMIT

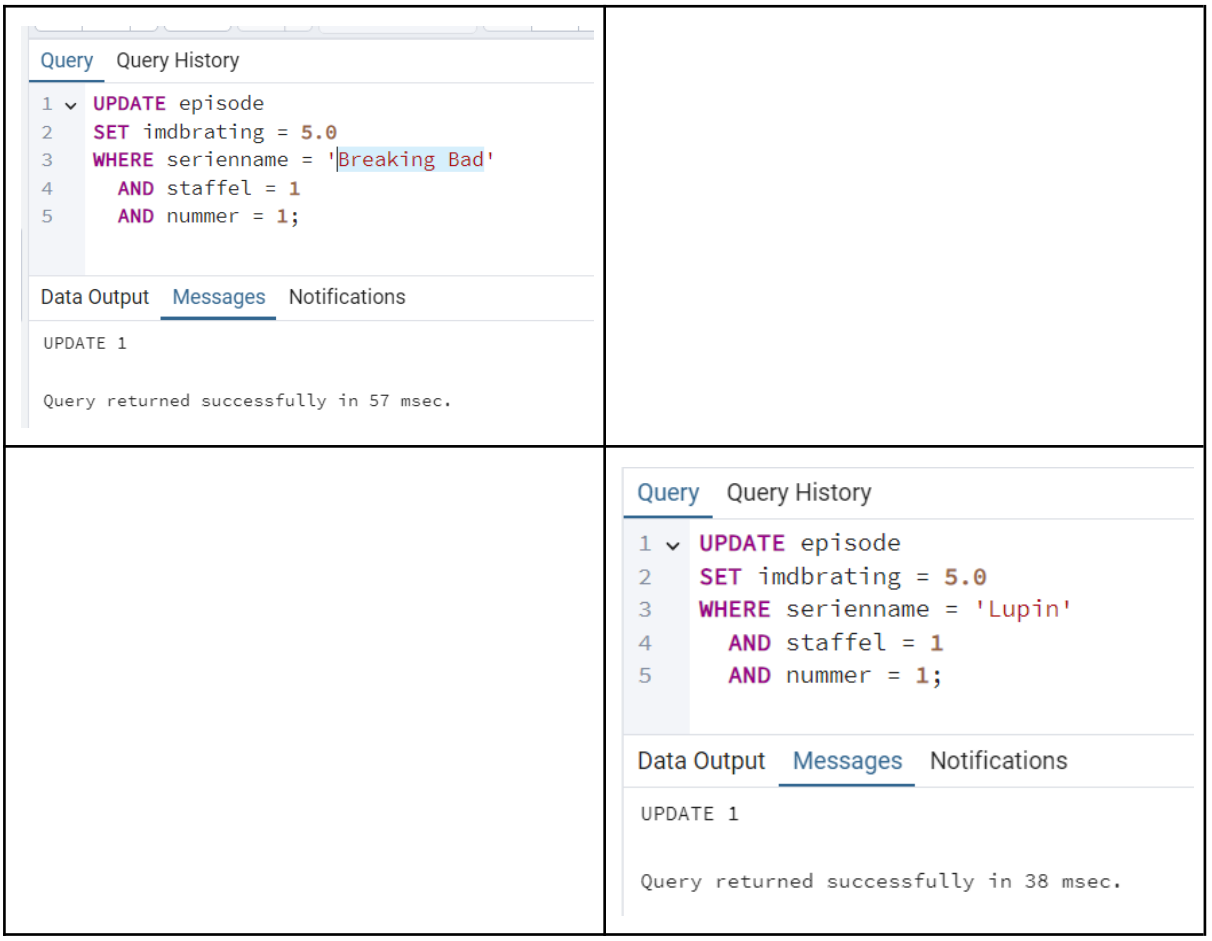
- Nach dem T1 Read wurde T2 gestartet welche einen WRITE gemacht hat
- Nach dem WRITE und Commit von T2, war die Ressource überschrieben.
- In T1 wurde der neue Wert gelesen => kein non-repeatable READ verhindert, es wurde der neue Wert von T2 WRITE ausgegeben
  T1 kann Änderungen sehen, die nach dem ersten Lesen gemacht wurden, solange sie committed sind.

- T1 kann mit WRITE den neuen Wert wieder überschreiben
- T1 READ liefert neuen Wert aus vorherigen WRITE
- Nach COMMIT von T1 wurde ein COMMIT ausgeführt.

Isolation Level (SQL 92)

Isolation Level	Dirty Read Daten werden vor einem commit / rollback gelesen	Nonrepeatable Read update / delete „von der Seite" mit commit möglich	Phantom Read insert „von der Seite" mit commit möglich
Read Uncommitted	möglich	möglich	möglich
Read Committed	**nicht möglich**	möglich	möglich
Repeatable Read	**nicht möglich**	**nicht möglich**	möglich
Serializable	**nicht möglich**	**nicht möglich**	**nicht möglich**

6. Deadlock:



<table>
<tr>
<td>

Query Query History

```
1 UPDATE episode
2 SET imdbrating = 8.0
3 WHERE serienname = 'Lupin'
4   AND staffel = 1
5   AND nummer = 1;
```

Data Output Messages Notifications

Total rows: 1 of 1 Waiting for the query to complete... 00:00:01.621 Ln 2, Col 19

</td>
<td></td>
</tr>
<tr>
<td></td>
<td>

Query Query History

```
1 UPDATE episode
2 SET imdbrating = 4.0
3 WHERE serienname = 'Breaking Bad'
4   AND staffel = 1
5   AND nummer = 1;
```

Data Output Messages Notifications

```
ERROR:  Process 41496 waits for ShareLock on transaction 3058084; blocked by process 39350.
Process 39350 waits for ShareLock on transaction 3058085; blocked by process 41496.deadlock detected

ERROR:  deadlock detected
SQL state: 40P01
Detail: Process 41496 waits for ShareLock on transaction 3058084; blocked by process 39350.
Process 39350 waits for ShareLock on transaction 3058085; blocked by process 41496.
Hint: See server log for query details.
Context: while updating tuple (1,44) in relation "episode"
```

</td>
</tr>
<tr>
<td>Transaktion ausgeführt von T1, nachdem der Deadlock von T2 erkannt wurde</td>
<td>Transaktion aborted, Rollback notwendig</td>
</tr>
</table>

Nur Tupel gesperrt