

Lecture 12

- Transactions
- Locking
- Dirty reads etc.

Make it or break it – breaking it

- Sometimes, we need several SQL statements in sequence to succeed or fail as a unit
- Consider, e.g., posting an image in our social network database. This requires us to insert both into the **Image** and the **Image_owner** table.
- Now consider what happens if, between the two queries, another process deletes the user from the **User** table that was going to be the image owner.
- Then the INSERT into the **Image_owner** table fails because the foreign key constraint on the **owner_id** is not met
- Result: We have an image that has no owner. This is undesirable.

Making it – with transactions

- Many SQL RDBMS allow us to use *transactions* to let SQL data manipulation statements (SELECT, INSERT, UPDATE, DELETE) succeed or fail as a unit
- If all statements in a transaction succeed, you can COMMIT (end) the transaction. This makes all changes permanent.
- You can also roll back statements manually with ROLLBACK.

Transaction – simple example

```
START TRANSACTION;
INSERT INTO Image(`image_data`,
                 `caption`,
                 `is_public`)
VALUES ('FFD8...', 'My mate!', 1);
INSERT INTO Image_owner (`image_id`,
                        `owner_id`)
VALUES (LAST_INSERT_ID(), 555);
```

- Followed by
COMMIT;
or
ROLLBACK;

What happens in this transaction?

- If there is a `user_id` in the `User` table that has the value 555, the foreign key constraint for `owner_id` in `Image_owner` is satisfied and both INSERT queries succeed.
- If there is no such `user_id` in the `User` table, the foreign key constraint is not satisfied and the INSERT into `Image_owner` fails.
- If we now issue a COMMIT, this means that the change to one or both tables becomes permanent.
- If we issue a ROLLBACK, the transaction is rolled back, i.e., neither of the two INSERT statements has a permanent effect.
- Note that in some clients, such as PHPMyAdmin, a transaction is rolled back as soon as an error is discovered. In other clients, such as the mysql command line client, we need to discover ourselves whether an error has occurred and need to decide whether to COMMIT or ROLLBACK

Multiple users (clients)

- “Too many cooks spoil the porridge”

Client 1 (store cash register application):

```
SELECT `Items in stock`  
FROM stock  
WHERE  
  `Article number` = '654-6789'  
INTO @items;
```

```
UPDATE stock  
SET `Items in stock` =  
  @items - 1  
WHERE  
  `Article number` = '654-6789'
```

Client 2 (online store application):

```
SELECT `Items in stock`  
FROM stock  
WHERE  
  `Article number` = '654-6789'  
INTO @items;
```

```
UPDATE stock  
SET `Items in stock` =  
  @items - 1  
WHERE  
  `Article number` = '654-6789'
```

time

How many items do we have in stock, and how many does the stock table say we have?

What's the problem?

- Client 2 updates the stock table after Client 1 has read the stock level, but before Client 1 has had a chance to update it to record the sale
- Client 1 then updates based on outdated information on number of items in stock

Could we fix this with a transaction?

- Not really, because neither client is aware of the other's actions
- What we need is to be able to reserve tables for the use of a particular client
- This is called a *lock*

Locking and unlocking tables

- To lock a table for reading (=nobody can write to it until the table is unlocked):

```
LOCK TABLES <table_name> READ;
```

- To lock a table for writing (=nobody can access the table for read or write):

```
LOCK TABLES <table_name> WRITE;
```

- To release a lock:

```
UNLOCK TABLES;
```

Multiple users (clients) with write locks

Client 1 (store cash register application):

```
LOCK TABLES stock WRITE;
SELECT `Items in stock` FROM stock
WHERE `Article number` = '654-6789'
INTO @items;
UPDATE stock
SET `Items in stock` = @items - 1
WHERE `Article number` = '654-6789';
UNLOCK TABLES;
```

Client 2 (online store application):

```
LOCK TABLES stock WRITE;
Waiting to be able
to acquire a lock
on the table
SELECT `Items in stock` FROM stock
WHERE `Article number` = '654-6789'
INTO @items;
UPDATE stock
SET `Items in stock` = @items - 1
WHERE `Article number` = '654-6789';
UNLOCK TABLES;
```

time

How many items do we have in stock, and how many does the stock table say we have?

Transaction problems

- A fundamental problem with transactions is that writes (INSERTs, UPDATEs, or DELETEs) to tables do not become permanent until the transaction is committed
- However, other sessions may attempt to read data from the tables with SELECT while the transaction is underway
- A number of undesirable conditions may arise: *dirty reads*, *non-repeatable reads*, and *phantom reads*
- The *transaction isolation level* at which all of these problems can arise is called “read uncommitted”

Dirty reads

- Consider a transaction that has just carried out an INSERT, UPDATE or DELETE statement but that has not committed yet.
- Meanwhile, another session does a SELECT on the table. The result set includes the uncommitted changes to the table that were made by the incomplete transaction.
- This transaction is now rolled back. So the result set included changes that were never meant to happen.
- This is a *dirty read*. To prevent a dirty read, SELECT statements must be prevented from returning uncommitted changes in result sets.
- This is achieved with a transaction isolation level of “read committed”

Non-repeatable reads

- Consider a transaction that carries out two SELECT queries on the same table some time apart
- Between the two SELECTs, another session writes to the table, generally via an UPDATE (and commits if part of a transaction)
- On the second select query, the transaction sees different data in the record(s) of the result set because the data in the table has been changed.
- This is called a *non-repeatable read*. To prevent a non-repeatable read, the transaction must either lock the table rows it wishes to read, or it must work on a copy (snapshot) of the data as it existed at the beginning of the transaction.
- The transaction isolation level “repeatable read” prevents non-repeatable reads.

Phantom reads

- The scenario in a phantom read is essentially the same as in a non-repeatable read, except that the change in the data is brought about by an INSERT rather than an UPDATE
- That is, the second SELECT sees *new* data, not just modified data
- This cannot be prevented by merely placing read locks on the existing records, because new records are added
- Avoiding phantom reads all together requires an isolation level of “serializable”. Simplified, this means that each transaction locks all tables it deals with for reading and/or writing, and no two transactions execute in parallel.

Isolation levels

- Conflicts due to insufficient isolation levels are relatively rare in practice
- However, enforcing isolation levels is computationally expensive (locking and unlocking as well as waiting for lock releases or commits is time-consuming)
- For this reason, many RDBMS do not operate with “serializable” as isolation level
- E.g., MySQL comes out of the box with “repeatable read” as default

Today’s lab sheet

- ...is on the web
- Today: locking and unlocking tables, transactions – committing and rolling back, working at isolation level *repeatable read*