

# Tips and Tricks for Project Task 3

## Data Storage Paradigms, IV1351

### 1 Lock for (no key) update When Required, Important for Both Task A and B

There are situations where the *lost update* anomaly may appear, even though transactions are used correctly. This section explains that risk, and how to avoid it. First, let's look at the anomaly using a classical bank account example. Say that two users, U1 and U2, are about to withdraw 100 SEK from the same bank account, which has the balance 300 SEK. They use different ATMs, and both ATMs call the bank's server. At the server, the requests for withdrawal run in parallel, U1's withdrawal is handled by transaction Tx1 and U2's by Tx2. The schedule we want to avoid is illustrated in Table 1.

Operation no	Tx1	Tx2
1	read balance 300	
2	calculate new balance = 300 - 100 = 200	
3		read balance 300
4		calculate new balance = 300 - 100 = 200
5	write balance 200	
6	commit	
7		write balance 200
8		commit

Table 1: An example of the lost update anomaly. Two users withdraw 100 SEK each, but only 100 SEK in total is reduced from the account's balance.

First of all, the transactions Tx1 and Tx2 must span a complete withdrawal. It would be wrong to have separate transactions for reading, operations one and three in Table 1, and writing, operations five and seven. This might seem obvious, but can in fact easily be overlooked. The reason is that the calculations, operations two and four, are business rules that are often handled in the program, not in database calls. Also, when using

a layered architecture, which is often the case, the business rules aren't handled in the same layer as the database calls. In this situation, it's quite easy to make the erroneous implementation explained in Figure 1.

1. Call the database layer to read the balance. This is done in one transaction that commits when reading is completed.
2. Call the business logic layer to manage all business rules related to a withdrawal, including calculating the new balance.
3. Call the database layer to write the updated balance, which is then done in another transaction than that used in bullet 1 above.

*Figure 1: Erroneously reading from the database in one transaction, and writing an updated result in another transaction*

However, depending on transaction isolation level and concurrency control mechanism, the lost update anomaly illustrated in Table 1 can happen even if we *don't* make the mistake in Figure 1. Without going into technical details, let's just state that the default isolation level of MySQL is Repeatable Read, and for Postgres it is Read Committed, and that they both use variants of MVCC concurrency control. The lost update anomaly can occur in both those cases (and for most other DBMSs as well). As a side note, it might seem strange that the lost update anomaly can occur in the Repeatable Read isolation level in MySQL. The reason is that MySQL uses MVCC, while it's only when using 2PL that lost updates are impossible in Repeatable Read isolation. Postgres also uses a variant of MVCC, but adds extra checks that prohibits lost updates in Repeatable Read.

Anyway, a straightforward way to avoid the lost update in Table 1 is to acquire an exclusive lock already when reading the data, in order to block other transactions from reading the same data. This exclusive lock is acquired by adding a locking clause to the SELECT statement, as in Listing 1. This method to avoid lost updates has the advantage that it works in all DBMSs.

```
1 SELECT balance
2 FROM account
3 WHERE account_id = 12345
4 FOR UPDATE
```

*Listing 1: Using a locking clause, in bold, to make a select statement acquire an exclusive lock.*

The FOR UPDATE lock clause in Listing 1 is standard SQL, but most databases have their own additions regarding locks. Postgres has FOR NO KEY UPDATE, see Listing 2, which doesn't lock the PK column. This still solves our lost update problem discussed here, but allows other transactions to insert rows in another table, which has a FK referencing the PK in the row we have locked. This improves performance, and doesn't



introduce any problem as long as the PK column isn't updated.

```
1 SELECT balance
2 FROM account
3 WHERE account_id = 12345
4 FOR NO KEY UPDATE
```

*Listing 2: This locking clause doesn't lock the PK column of the locked row.*

## 2 Avoid Business Logic in DAOs, Important Only for Task B

There shall not be any business logic at all in classes responsible for calling the databases, since we use the *Layer* pattern. That pattern states that each layer shall have just one, clearly defined responsibility. The responsibility of the *integration* layer, from where the database calls are made, is to call external systems. To also include business logic in the same layer would give the classes in that layer very bad cohesion, or in other words, they would do a bit of everything, thereby being hard to understand and maintain. Also the *MVC* pattern tells us not to have business logic in the integration layer, since it states that all business logic shall be placed in the *model* layer. This is for the same reason, to achieve high cohesion, having just one responsibility for each class and layer.

### A Definition of Business Logic and Business Rules

To be able to avoid business logic in the integration layer, it must be first be understood exactly what the term *business logic* actually means. The answer is that it's the code implementing business rules, which requires a definition also of the term *business rule*. The business rules define how data is allowed to be shown, deleted, created, or altered, and are rules that would apply even if there were no computers, programs or databases at all. As an example, we can consider rules for withdrawals from a bank account. A perhaps obvious rule is that the amount of money handed to the customer must be subtracted from the account balance. Other rules could be that it's not possible to withdraw more money than there's in the account, that it's not possible to withdraw a negative amount, and that only the account owner can make a withdrawal.

### How to Avoid Business Logic in DAOs

It's not possible to state a method that's guaranteed to remove all business logic from the integration layer. The only way to make sure that's done is to check all code in that layer, and make sure it doesn't handle any business rules, according to the definition of business rules above. There's however a very useful best practice, which aims to prohibit business logic in DAO methods by restricting allowed method names. The idea is that the allowed method names shall only be appropriate for methods that are "dumb", from a business point of view. They create, read, update or delete data, without bothering



about business rules. Such methods are often called CRUD operations (**C**reate, **R**ead, **U**ppdate, **D**elete). The allowed method names in DAOs are specified in the list below.

- `create<something>`, for example `createAccount`, which stores an instance of `<something>` in the database.
- `find<something>`, for example `findProducts`, returning all instances of `<something>`. There might also be methods called `find<something>By<Criteria>`, like `findProductsByCategory`, which returns all instances of `<something>` matching `<Criteria>`. The method `findProductsByCategory` would return all products belonging to the specified category.
- `update<something>`, for example `updateAccount`, which updates the state of `<something>` in the database.
- `delete<something>`, for example `deleteAccount`, which deletes an instance of `<something>` from the database. There may also be methods deleting multiple instances, called `delete<something>By<Criteria>`, for example `deleteAccountByHolder`, which deletes all instances of `<something>` that matches `<Criteria>`. The method `deleteAccountByHolder` would take an account holder as parameter, and delete all accounts owned by that holder.

This naming restriction prohibits creating methods called for example `withdraw` or `rent`, which is very useful since such operations tend to involve business logic. It's however important to understand that this practice is just a tool that helps, it's not a sufficient condition guaranteeing there's no business logic. The problem is that the whole idea of this guideline can be circumvented, for example by creating a method called `createWithdrawal`, which contains all business rules related to withdrawing money from a bank account!

### 3 Strive for an Object-Oriented Public Interface in DAOs, Important Only for Task B

It's best to use objects as method parameters and return values in DAOs, provided the system is written in an object-oriented language. Using primitive values would only require translation between primitive values and objects somewhere else in the program, or, in worst case, lead to the use of primitive values instead of objects in the entire program.

Strictly following this practice does, however, have an important disadvantage, it becomes very difficult to do aggregations in DAOs. Say for example that it's required to know the number of accounts in a bank. This is very easy to do in SQL, with a statement like `SELECT COUNT(*) FROM account`. The DAO method containing this SQL would then simply return the number of rows, but the problem is that this number is a primitive value. If we were really strict on only returning objects, the DAO would have to read all rows from the `account` table, create an object of an `Account` class for each row, and return a list containing all those `Account` objects. The class calling the DAO would then get the number of accounts in the bank by getting the number of



elements in that list. That would be an awful lot of unnecessary and time-consuming extra work. The only reasonable solution is to be a bit relaxed on the object-oriented practice, and do exactly what the title of this section says, *strive* for an object-oriented public interface in DAOs, but realise that it's not always reasonable.

Whether to do aggregations in a DAO, or not, is just one case of a bigger discussion, how to divide work between the application and the database. At one extreme, the database does absolutely nothing except to store data. All constraints on the data are checked in the application, and all kinds of aggregations and calculations are also done in the application. On the other extreme, the entire application logic can in fact be written in the database, since it's possible to write programs in all major DBMSs. Postgres, for example, can run programs written in for example C, Python and the Postgres-specific language PL/pgSQL. However, how to divide tasks between application and database will not be discussed here, let's just conclude that it depends on many factors, like cost, code maintainability, execution speed, developer knowledge, and how existing code is written. What matters most is, as always, to make an informed decision, based on facts.