Explanation and Model Solution for Example Assignment: Databases—Inside the Black Box

COMPSCI 2DB3: Databases

Jelle Hellings Holly Koponen

Department of Computing and Software
McMaster University

Foreword

As the first step in this model solution, we walk over the description. Most parts of the description are relevant, but a few parts can be omitted. Then, we provide a general analysis highlighting important aspects of the working of the proposed approach. Finally, we answer each of the evaluation questions.

Model Solution

~~A financial service provider came to the conclusion that their system, which processes complex money transfers between people, is not fast enough. Hence, they asked a consultant to come up with a higher-performance system. The consultant took a look at the original system and concluded that *lock-based* access was the main culprit. To deal with these performance issues, the consultant came up with a *novel* design that reduces the duration of locks. Next, we detail the proposed design.~~

In the system, all transactions can be written as a sequence of *transfers* of the form

$$\underline{\text{transfer}}(\$x, \mathit{from}, \mathit{to}) = \text{“transfer \$x from account } \mathit{from} \text{ to account } \mathit{to}\text{”},$$

that should only be executed if *each transfer* is possible (the *from*-account has sufficient funds). E.g., the transaction

$$\tau = [\underline{\text{transfer}}(\$500, \mathit{Bo}, \mathit{Alicia}), \underline{\text{transfer}}(\$300, \mathit{Eva}, \mathit{Celeste})]$$

of two such transfers is equivalent to

“if *Bo* has at-least \$500 and *Eva* has at-least \$300,
then transfer \$500 from *Bo* to *Alicia* and transfer \$300 from *Eva* to *Celeste*”.

~~The consultant wants to execute these transactions with a minimal amount of locking.~~ To do so, the consultant designed the minimal-locking operations UPDATE-BALANCE and TAKE-BALANCE-CONDITIONAL (see Figure 1 for the pseudo-code of these operations). Using these operations, the consultant proposes to execute transactions τ with n *transfers*, e.g., $\tau = [\underline{\text{transfer}}(\$x_1, \mathit{from}_1, \mathit{to}_1), \dots, \underline{\text{transfer}}(\$x_n, \mathit{from}_n, \mathit{to}_n)]$, using the EXECUTE-TRANSACTION algorithm (see Figure 2 for the pseudo-code of this algorithm). The EXECUTE-TRANSACTION algorithm will visit each *from*-account, check whether that account has sufficient funds (at-least the funds required for the transfer), and take away the funds that are to-be transferred (a *reservation of funds*). This reservation can be used in two ways:

1. If all *from*-accounts have sufficient funds, then all reserved funds will be transferred to their respective *to*-accounts (the transaction is successful). To do so, the variable *Commit* lists all UPDATE-BALANCE operations necessary to transfer reserved funds to their respective *to*-accounts.

```
UPDATE-BALANCE(τ, account, amount):
Lock_τ(account).
account := account + amount.
Release_τ(account).

TAKE-BALANCE-CONDITIONAL(τ, account, amount):
Lock_τ(account).
if account ≥ amount then
    account := account − amount.
    Release_τ(account).
    return True.
else
    Release_τ(account).
    return False.
end if
```

Figure 1: The pseudo-code for the minimal-locking operations UPDATE-BALANCE and TAKE-BALANCE-CONDITIONAL.

2. Otherwise, if a *from*-account is found without sufficient funds, then all previously reserved funds will be returned to their respective *from*-accounts (the transaction failed). To do so, the variable *Rollback* lists all UPDATE-BALANCE operations necessary to transfer reserved funds back to their respective *from*-accounts.

~~The consultant believes that this setup will reduce locking, but might introduce unwanted interference between transactions.~~ The financial service provider has already been instructed about the risks of interference, and decided that it can agree to interference as long as the following *constraints* are never broken:

C1. No account should ever receive a negative balance (assuming that all accounts start with a positive balance).

C2. As the transfers only move money between accounts, no money should be *lost* or *created*. Hence, if at any time t no transactions are being executed, then the sum of the balances of all accounts at that time t should be equivalent to the initial sum of the balances of all accounts.

C3. Successful transactions must have their *lasting effects*, while failed transactions must not have lasting effects. Hence, if at any time t no transactions are being executed, then the balance of each account should reflect the balance updates due to all transactions that executed successfully before t.

We note that these constraints do not rule out *inconsistencies* in the data while transactions are being executed.

Faced with the complexity of the approach proposed by the consultant, the financial service provider has contacted you to evaluate the proposed approach.

> **Analysis.** Before we look at the evaluation questions, we interpret the above description using example transactions. If we look at concurrency issues, we should consider the interleaved interaction of *several* transactions that each affect the same database objects. Such transactions are possible in the limited form of transactions described. E.g.,
>
> $$\tau_1 = [\underline{\text{transfer}}(\$500, \mathit{Alicia}, \mathit{Bo}); \underline{\text{transfer}}(\$400, \mathit{Eva}, \mathit{Celeste})];$$
> $$\tau_2 = [\underline{\text{transfer}}(\$300, \mathit{Alicia}, \mathit{Dafni}); \underline{\text{transfer}}(\$200, \mathit{Celeste}, \mathit{Frieda})].$$

```
EXECUTE-TRANSACTION(τ):
Commit, Rollback := ∅, ∅.
for each transfer($x, from, to) in τ do
    if TAKE-BALANCE-CONDITIONAL(τ, from, $x) then
        Store the operation "UPDATE-BALANCE(τ, to, $x)" in Commit.
        Store the operation "UPDATE-BALANCE(τ, from, $x)" in Rollback.
    else
        Perform all operations in Rollback.
        return failure.
    end if
end for
Perform all operations in Commit.
return success.
```

Figure 2: The pseudo-code for the transaction execution algorithm.

Let us assume that the initial state of our database is given by Figure 3, *right*. If we look at the execution of τ_1 via EXECUTE-TRANSACTION(τ_1) in isolation on this database, then it will perform the operations outlined in Figure 3, *left*. Next, consider the execution of τ_2 via EXECUTE-TRANSACTION(τ_2) *at the same time* such that τ_2 starts executing *right after* the transfer transfer(\$500, *Alicia*, *Bo*) of τ_1 is fully processed (right after Line 7 in Figure 1, *left*). At this point, the balance of Alicia is updated to \$100. Hence, the transfer transfer(\$400, *Alicia*, *Dafni*) of τ_2 will fail directly and τ_2 will not make any changes to the database.

As is clear from the steps described in Figure 3, *left*, the locking protocol used does *not adhere* to two-phase locking or strict two-phase locking: after the lock on *Alicia* is released, the same transaction will obtain a lock on *Eva*.

Notice that each change made by TAKE-BALANCE-CONDITIONAL(τ, *from*, \$$x$) is a *removal* of \$$x$ amount of balance. Such a *removal* only happens if the balance is sufficient (at-least-\$$x$), as checked by the **if**-condition at Line 5 of Figure 1. No other transactions can interfere with this check-and-removal, as the check-and-removal happen while the account *from* is locked.

Second, we notice that for each such removal, we either execute a "UPDATE-BALANCE(τ, *to*, \$$x$)" (via *Commit*) that adds \$$x$ to the target account of the transfer or we execute a "UPDATE-BALANCE(τ, *from*, \$$x$)" (via *Rollback*) that undoes this *removal* of \$$x$ balance. In either case, the temporarily-removed balance is added back into the system before the end of the transaction.

Your evaluation

To evaluate the approach, the financial service provider asked you to investigate and answer the following questions:

1. Does the proposed approach follow strict two-phase locking? Does the proposed approach follow two-phase locking? Explain your answer. E.g., if the approach does not follow (strict) two-phase locking, then provide a transaction, its execution schedule, and argue that this schedule does not follow the (strict) two-phase locking protocol.

   Solution:

   The proposed approach does *not* follow strict two-phase locking and also does *not* follow two-phase locking. To see this, we consider the execution of a transaction

$$\tau_1 = \underline{\text{transfer}}(\$500, \mathit{Alicia}, \mathit{Bo}); \underline{\text{transfer}}(\$400, \mathit{Eva}, \mathit{Celeste})$$

1: -- **for**-loop of Line 2 with <u>transfer</u>(\$500, *Alicia*, *Bo*).
2: $\text{Lock}_{\tau_1}(\textit{Alicia})$. -- TAKE-BALANCE-CONDITIONAL(τ_1, *Alicia*, \$500).
3: $\text{Read}_{\tau_1}(\textit{Alicia})$. -- **if** *account* $\geq$ amount.
4: $\text{Write}_{\tau_1}(\textit{Alicia})$. -- *account* := *account* − amount.
5: $\text{Release}_{\tau_1}(\textit{Alicia})$. -- At Line 8 of Figure 1.
6: -- Store the operation "UPDATE-BALANCE(τ_1, *Bo*, \$500)" in *Commit*.
7: -- Store the operation "UPDATE-BALANCE(τ_1, *Alicia*, \$500)" in *Rollback*.
8: -- **for**-loop of Line 2 with <u>transfer</u>(\$400, *Eva*, *Celeste*).
9: $\text{Lock}_{\tau_1}(\textit{Eva})$. -- TAKE-BALANCE-CONDITIONAL(τ_1, *Eva*, \$400).
10: $\text{Read}_{\tau_1}(\textit{Eva})$. -- **if** *account* $\geq$ amount.
11: $\text{Release}_{\tau_1}(\textit{Eva})$. -- At Line 10 of Figure 1.
12: -- Perform all operations in *Rollback* at Line 7.
13: $\text{Lock}_{\tau_1}(\textit{Alicia})$. -- UPDATE-BALANCE($\tau$, *Alicia*, \$500).
14: $\text{Read}_{\tau_1}(\textit{Alicia})$. -- *account* := *account* + amount.
15: $\text{Write}_{\tau_1}(\textit{Alicia})$. -- *account* := *account* + amount.
16: $\text{Release}_{\tau_1}(\textit{Alicia})$. -- At Line 3 of Figure 1.

Alicia	\$500
Eva	\$300
Bo	\$100
Celeste	\$300

Figure 3: The basic operations performed by transaction τ_1 (*left*) when executed on the provided database state (*right*).

on the database of Figure 3, *right*. If we look at the execution of τ_1 via EXECUTE-TRANSACTION(τ_1) in isolation on this database, then it will perform the operations outlined in Figure 3, *left*. As is clear from the steps described in Figure 3, *left*, τ_1 obtains a lock after it already released locks (namely τ_1 obtains a lock on *Eva* after releasing a lock on *Alicia*). This is in violation of the restrictions placed on executions of transactions by the two-phase locking and strict two-phase locking protocols.

2. Does the proposed approach suffer from *deadlocks*? Explain your answer.

Solution:

A *deadlock* can only occur when transactions try to obtain a lock while already holding a lock. E.g., if we start with a situation in which transactions τ and τ' hold locks on data objects O and O', respectively, after which τ and τ' try to obtain locks on O' and O, respectively, then τ will wait on τ' to release the lock on O' while, at the same time, τ' is waiting for τ to release the lock on O.

In the proposed approach, no transaction will obtain a lock while holding a lock. Hence, no deadlocks can occur.

3. Does the proposed approach expose *read-write*, *write-read*, or *write-write* conflicts? Explain your answer. E.g., if there are conflicts, then provide two transactions, a valid interleaved execution schedule for these transactions, and argue that this schedule has these conflicts.

Solution:

To show that the approach has *read-write* conflicts and *write-read* conflicts, we consider the following two transactions:

$$\tau_1 = [\underline{\text{transfer}}(\$400, \textit{Alicia}, \textit{Bo}), \underline{\text{transfer}}(\$300, \textit{Celeste}, \textit{Dafni})];$$
$$\tau_2 = [\underline{\text{transfer}}(\$200, \textit{Alicia}, \textit{Celeste})],$$

and we consider the following valid interleaved execution of τ_1 and τ_2 that follows from the proposed approach (we use TBC as a shorthand for TAKE-BALANCE-CONDITIONAL and UB as a shorthand for UPDATE-BALANCE):

Alicia	\$500
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\textbf{return}\ \texttt{True}]{\text{TBC}(\tau_1,\ \mathit{Alicia},\ \$400)}$

Alicia	\$100
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\textbf{return}\ \texttt{False}]{\text{TBC}(\tau_2,\ \mathit{Alicia},\ \$200)}$

Alicia	\$100
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\textbf{return}\ \texttt{False}]{\text{TBC}(\tau_1,\ \mathit{Celeste},\ \$300)}$

Alicia	\$100
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\text{via}\ \mathit{Rollback}]{\text{UB}(\tau_1,\ \mathit{Alicia},\ \$400)}$

Alicia	\$500
Bo	\$100
Celeste	\$100
Dafni	\$100

.

This interleaved execution yields the schedule

τ_1	τ_2
$\text{Lock}_{\tau_1}(\mathit{Alicia})$	
$\text{Read}_{\tau_1}(\mathit{Alicia})$	
$\text{Write}_{\tau_1}(\mathit{Alicia})$	
$\text{Release}_{\tau_1}(\mathit{Alicia})$	
	$\text{Lock}_{\tau_2}(\mathit{Alicia})$
	$\text{Read}_{\tau_2}(\mathit{Alicia})$
	$\text{Release}_{\tau_1}(\mathit{Alicia})$
$\text{Lock}_{\tau_1}(\mathit{Celeste})$	
$\text{Read}_{\tau_1}(\mathit{Celeste})$	
$\text{Release}_{\tau_1}(\mathit{Celeste})$	
$\text{Lock}_{\tau_1}(\mathit{Alicia})$	
$\text{Read}_{\tau_1}(\mathit{Alicia})$	
$\text{Write}_{\tau_1}(\mathit{Alicia})$	
$\text{Release}_{\tau_1}(\mathit{Alicia})$	
Abort_{τ_1}	
	Abort_{τ_2}

In this execution, τ_2 reads an uncommitted value from *Alicia* that was previously written by τ_1 (a write-read conflict) and τ_1 writes a value to *Alicia* that was previously read by τ_2 (a read-write conflict).

To show that the approach has *write-write* conflicts, we consider the following valid interleaved execution of τ_1 and τ_2 that follows from the proposed approach (we use TBC as a shorthand for

TAKE-BALANCE-CONDITIONAL and UB as a shorthand for UPDATE-BALANCE):

Alicia	\$800
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\textbf{return}\ \texttt{True}]{\text{TBC}(\tau_1,\ \textit{Alicia},\ \$400)}$

Alicia	\$400
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\textbf{return}\ \texttt{True}]{\text{TBC}(\tau_2,\ \textit{Alicia},\ \$200)}$

Alicia	\$200
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\textbf{return}\ \texttt{False}]{\text{TBC}(\tau_1,\ \textit{Celeste},\ \$300)}$

Alicia	\$100
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\text{via } \textit{Rollback}]{\text{UB}(\tau_1,\ \textit{Alicia},\ \$400)}$

Alicia	\$600
Bo	\$100
Celeste	\$100
Dafni	\$100

$\xrightarrow[\text{via } \textit{Commit}]{\text{UB}(\tau_2,\ \textit{Celeste},\ \$200)}$

Alicia	\$600
Bo	\$100
Celeste	\$300
Dafni	\$100

.

We note that we used a *different* initial database state in this execution. This interleaved execution yields the schedule

τ_1	τ_2
$\text{Lock}_{\tau_1}(\textit{Alicia})$	
$\text{Read}_{\tau_1}(\textit{Alicia})$	
$\text{Write}_{\tau_1}(\textit{Alicia})$	
$\text{Release}_{\tau_1}(\textit{Alicia})$	
	$\text{Lock}_{\tau_2}(\textit{Alicia})$
	$\text{Read}_{\tau_2}(\textit{Alicia})$
	$\text{Write}_{\tau_2}(\textit{Alicia})$
	$\text{Release}_{\tau_1}(\textit{Alicia})$
$\text{Lock}_{\tau_1}(\textit{Celeste})$	
$\text{Read}_{\tau_1}(\textit{Celeste})$	
$\text{Release}_{\tau_1}(\textit{Celeste})$	
$\text{Lock}_{\tau_1}(\textit{Alicia})$	
$\text{Read}_{\tau_1}(\textit{Alicia})$	
$\text{Write}_{\tau_1}(\textit{Alicia})$	
$\text{Release}_{\tau_1}(\textit{Alicia})$	
	$\text{Lock}_{\tau_2}(\textit{Celeste})$
	$\text{Read}_{\tau_2}(\textit{Celeste})$
	$\text{Write}_{\tau_2}(\textit{Celeste})$
	$\text{Release}_{\tau_1}(\textit{Celeste})$
Abort_{τ_1}	
	Commit_{τ_2}

In this execution, τ_2 *reads and writes* an uncommitted value from *Alicia* that was previously written by τ_1 (a write-write conflict), after which τ_1 *reads and writes* that same uncommitted value from *Alicia* just written by τ_2 (another write-write conflict).

4. Does the proposed approach suffer from *dirty reads*? Explain your answer. E.g., if there are dirty reads, then provide two transactions, a valid interleaved execution schedule for these transactions, and argue that this schedule has dirty reads.

   Solution:

   The approach has dirty reads. We refer to the transactions τ_1 and τ_2 used in the solution of

		First τ_1, then τ_2		First τ_2, then τ_1	
Alicia	\$500	*Alicia*	\$300	*Alicia*	\$300
Bo	\$100	*Bo*	\$100	*Bo*	\$100
Celeste	\$100	*Celeste*	\$300	*Celeste*	\$300
Dafni	\$100	*Dafni*	\$100	*Dafni*	\$100

Figure 4: An initial database state (*left*), the database state after executing the transactions τ_1, τ_2 from the solution of Evaluation Question 7, and the database state after executing the transactions τ_2, τ_1 from the solution of Evaluation Question 7.

Evaluation Question 3 and the first interleaved execution and schedule. In this execution, τ_2 reads the value \$100 from *Alicia*, which is an uncommitted value written earlier by τ_1.

5. Does the proposed approach suffer from *unrepeatable reads*? Explain your answer. E.g., if there are unrepeatable reads, then provide two transactions, a valid interleaved execution schedule for these transactions, and argue that this schedule has unrepeatable reads.

   Solution:

   The proposed approach suffers from unrepeatable reads. We refer to the transactions τ_1 and τ_2 used in the solution of Evaluation Question 3 and the second interleaved execution and schedule. In this schedule, τ_1 reads and writes the value \$400 to *Alicia*. On the subsequent read by τ_1, τ_1 reads the value \$200 from *Alicia*. Hence, this repeated read reads a different value than previously established by τ_1.

6. Is the proposed approach *serializable*? Explain your answer.

   Solution:

   Absolutely not: we have already seen that the proposed approach suffers from both dirty reads and unrepeatable reads.

   Alternatively, we shall show a transaction execution that is not equivalent to any serial execution. We refer to the transactions τ_1 and τ_2 used in the solution of Evaluation Question 3. The only two *serial* executions of τ_1 and τ_2 are first executing τ_1 entirely and then τ_2 or first executing τ_2 entirely and then executing τ_1. On the initial database state of Figure 4, *left*, this leads to the database states of Figure 4, *middle* and *right*, respectively.

   Now consider the interleaved execution of τ_1 and τ_2 presented in the solution of Question 3, which can follow from the proposed approach. As one can see, this execution is different from any *serial* execution. Hence, the proposed approach is not serializable.

7. Are the constraints C1-C3, as set out by the financial service provider, satisfied? Explain your answer. E.g., if a constraint is satisfied, then argue why that is the case.

   Solution:

   (C1) *No account should ever receive a negative balance (assuming that all accounts start with a positive balance).*
   This constraint *holds*. The balance of an account can only *become* negative if balance is removed. Hence, as long as balance is *only* removed if sufficient balance is available, no account will ever have a negative balance. We observe that only Line 6 of Figure 1 removes balance from some account *from*, and this removal is guarded by the check on the preceding line that verifies whether *from* has sufficient balance. No other transactions can interfere with this check-and-removal, as the check-and-removal happen while the account *from* is locked.

(C2) *As the transfers only move money between accounts, no money should be* lost *or* created. *Hence, if at any time t no transactions are being executed, then the sum of the balances of all accounts at that time t should be equivalent to the initial sum of the balances of all accounts.*
This constraint *holds*. Balance is only removed by TAKE-BALANCE-CONDITIONAL(τ, *from*, \$$x$), and TAKE-BALANCE-CONDITIONAL only does so if it returns `True`. In that case, a balance addition operation is added to both *Commit* and *Rollback* (namely, an operation "UPDATE-BALANCE(τ, *to*, \$$x$)" is added to *Commit*; and an operation "UPDATE-BALANCE(τ, *from*, \$$x$)" is added to *Rollback*). In all executions of EXECUTE-TRANSACTION, either only all operations collected in *Commit* (Line 11) are performed or only all operations collected in *Rollback* (Line 7) are performed. Hence, each removal of \$$x$ balance is accompanied by an addition of \$$x$ balance later on.

(C3) *Successful transactions must have their* lasting effects*, while failed transactions must not have lasting effects. Hence, if at any time t no transactions are being executed, then the balance of each account should reflect the balance updates due to all transactions that executed successfully before t.*
This constraint *holds*. Successful transactions perform balance removals of \$$x$ balance from accounts *from* via TAKE-BALANCE-CONDITIONAL operations and, for each such removal, perform a single balance addition of \$$x$ balance to some account *to* via an operation UPDATE-BALANCE collected in *Commit*. The combination of such a removal and addition operation corresponds to the execution of a single <u>transfer</u>(\$$x$, *from*, *to*) clause (a *lasting* change).
Failed transactions perform balance removals via TAKE-BALANCE-CONDITIONAL operations and, for each such removal, perform a rollback operation that restores the removed balance via an operation UPDATE-BALANCE collected in *Rollback*. The combination of such a removal and addition operation assures that no *lasting* balance is made by failed transactions.