

Trading, Banking, Ticket Systems (Part 2)

Big Data Engineering
(formerly Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

June 22, 2023

Cheat Sheet for Part 1

1. Transaction: bundles several SQL statements
2. ACID: Atomicity, Consistency, Isolation, Durability. For each transaction, the ACID properties must be guaranteed.
3. But how do we do that? In particular, how do we guarantee **automatic** isolation?
4. Schedule (History): concrete interleaving of read/write operations
5. Serial schedule: Transactions are executed one after the other
6. Conflict serialisable schedule: equivalent to a serial schedule
7. Conflict graph: aggregated form of a schedule: vertices correspond to transactions
8. No cycle in the conflict graph \Rightarrow associated schedule is conflict serialisable

Transactions in SQL (Transactions_DuckDB.ipynb)

Rollback Transactions

The next example shows a similar transaction as above. The only difference is that instead of making the changes persistent, we decide to `ABORT` the transaction by calling `rollback()` on the connection. It is equivalent to running the following SQL statements:

```
BEGIN;  
UPDATE accounts SET balance = balance - 100 WHERE id=3;  
UPDATE accounts SET balance = balance + 100 WHERE id=1;  
ABORT;
```

All changes performed by the aborted transaction must not become durable in the database. Note that if we `close()` an open connection, `rollback()` will be performed implicitly.

```
1 reset_database()  
2  
3 # establish two connections  
4 conn1 = duckdb.connect(database='accounts.duckdb')  
5 conn2 = duckdb.connect(database='accounts.duckdb')  
6  
7 # start a new transaction  
8 conn1.begin()  
9  
10 # update balance of account 3  
11 conn1.execute("""UPDATE accounts SET balance = balance - 100 WHERE id=3;""")  
12 # update balance of account 1  
13 conn1.execute("""UPDATE accounts SET balance = balance + 100 WHERE id=1;""")  
14 # compare states visible to both connections (transactions)  
15 q_acc = """SELECT * FROM accounts WHERE id=1 OR id=3;"""  
16 conn1.execute(q_acc)  
17 conn2.execute(q_acc)  
18 print(f"Account balances observed by each connection before COMMIT:\n"\n  
19       f"Transaction 1: {conn1.fetchall()}\n"\n  
20       f"Transaction 2: {conn2.fetchall()}\n"\n  
21       f"Changes not yet visible to connection 2.")  
22 )  
23  
24 # explicitly rollback the changes performed by the first connection  
25 conn1.rollback()  
26 print(f"---Transaction 1 aborted---")  
27
```

https://github.com/BigDataAnalyticsGroup/bigdataengineering/blob/master/Transactions_DuckDB.ipynb

Agenda: Possible Isolation Problems and Their Solutions

Problems:

1. Dirty Read
2. Non-Repeatable Read
3. Cascading Rollback

Solution: tuple-based locking

but how exactly? short-term, long-term, 2PL, S2PL?

However, **two other problems** are still possible:

4. Phantom Problem

Solution: predicate-based locking

5. Deadlock Problem

Solution: Wait-for-graph



phantom
problem

deadlocks

dirty read

Dirty Read

Dirty Read

We use **Dirty Read** to refer to the reading of a value by a transaction that was written by another uncommitted or aborted transaction. This means that a value was read that, in terms of isolation, should not yet have been visible to other transactions.

Example:

S: $w_1(A) \rightarrow \underbrace{r_2(A)}_{\text{dirty read}} \rightarrow w_2(B) \rightarrow \underbrace{r_2(B)}_{\text{no dirty read}} \rightarrow \underbrace{r_1(B)}_{\text{dirty read}} \rightarrow c_2 \rightarrow c_1$

How do we avoid dirty reads?

Non-Repeatable Read

Non-Repeatable Read

We use **Non-Repeatable Read** to refer to the phenomenon where a transaction repeatedly reads the same data object from the database, but may get back different values. This means that the transaction sees a changed value on repeated read, which should not be possible in terms of isolation, as changes by concurrently executed transactions (including committed transactions) should not be visible.

Example:

S: $r_1(A) \rightarrow w_2(B) \rightarrow \underbrace{r_1(A)}_{\text{repeatable read}} \rightarrow w_2(A) \rightarrow c \rightarrow \underbrace{r_1(A)}_{\text{non-repeatable read}} \rightarrow \dots$

How do we avoid non-repeatable reads?

Cascading Rollback

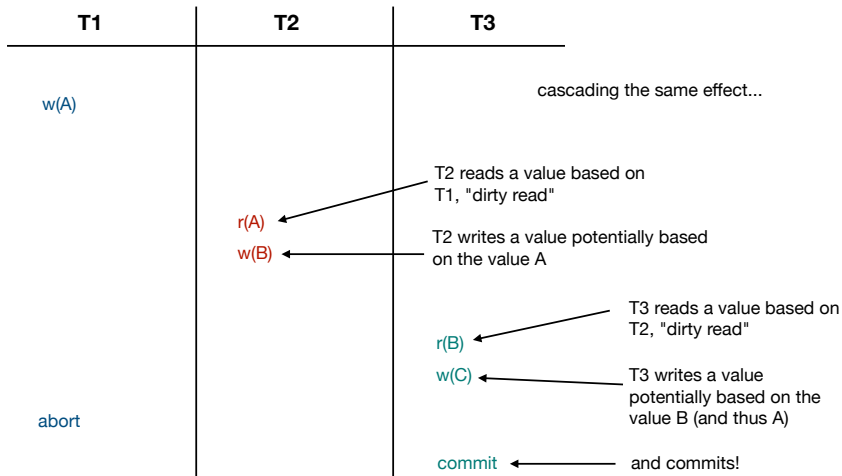
Cascading Rollback

We use **Cascading Rollback** to refer to the phenomenon of a transaction performing dirty reads and writing new values based on them. This means that the transaction performs new write operations based on non-committed values. This problem can lead to a whole cascade of transactions having to be reset.

Example:

S: $w_1(A) \rightarrow \underbrace{r_2(A)}_{\text{dirty read}} \rightarrow \underbrace{w_2(B)}_{\text{potentially invalid value}} \rightarrow a$

Extended Example for Cascading Rollback



=> Violation of A, C and I

How do we avoid cascading rollbacks?

Tuple-based Locking

Tuple-based Locking

1. Any operation that wants to read a tuple must first get a **read lock (shared lock, R or S)** for that tuple.
2. Any operation that wants to change a tuple must first get a **write lock (exclusive lock, X)** for that tuple.
3. If an operation does not receive a read or write lock, this operation must wait until it is allocated the lock.
4. All locks must be returned at the latest at the end of the transaction.

Notation:

We denote a read lock of transaction T_i on data object A_j with $getRLock_i(A_j)$ or $getSLock_i(A_j)$. We denote a write lock of transaction T_i on data object A_j with $getXLock_i(A_j)$. A lock is released with the operation $releaseLock_i(A_j)$.







Lock Compatibility

Read/write lock compatibility


For each tuple at any time:

1. If no write lock exists: any number of read locks are allowed,
2. If no read or write lock exists: one write lock is allowed.

When requesting locks, the following compatibility matrix must be followed:

		requested lock	
		S	X
existing lock	S		
	X		
	no		

 : lock is not granted => transaction must wait

 : lock is granted => transaction may continue

Short-Term Locks

Short-Term Locks

During a read operation $r_i^k(A_j)$, transaction T_i requests a lock for the tuple A_j immediately before the read operation in operation $k - 1$ and returns the lock immediately after the read operation ($k + 1$) (also for write operations).

Examples:

```
getSLock(A)  
read(A)  
releaseLock(A)
```

or:

```
getXLock(A)  
write(A, 100)  
releaseLock(A)
```

Long-Term Locks: Two-Phase-Locking (2PL)

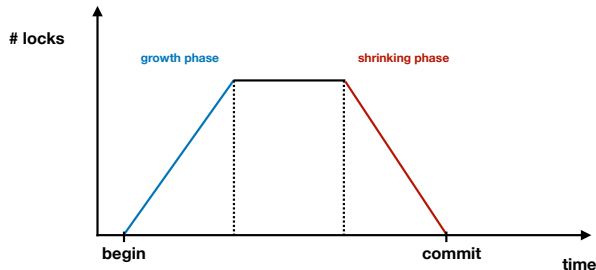
Two-Phase-Locking (2PL)

The duration of a transaction is divided into two phases:

1. **Growth phase:** The transaction can request locks.
2. **Shrinking phase:** The transaction can return locks.

As soon as a transaction returns the first lock, the shrinking phase begins.

Timeline for a transaction under 2PL:



Slide-in: 2PL vs Conflict Serialisability

2PL vs Conflict Serialisability

2PL only allows schedules that are conflict serialisable.

OK, then all is well, isn't it?

Unfortunately, no, because a schedule can also contain aborted transactions! In other words, any transaction can be aborted at any time by the user/database system.

Conflict serialisability only means that the schedule is equivalent to a serial schedule, but not that the transactions aborted in this schedule take back their introduced changes!

2PL does not solve this problem!

Strict Two-Phase-Locking (S2PL)

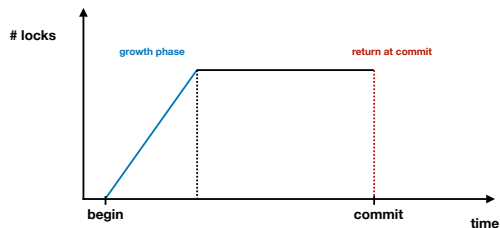
Strict Two-Phase-Locking (S2PL)

The duration of a transaction has only one phase:

1. **Growth phase:** For each operation $o_i^k(A_j)$, T_i must request the corresponding lock on A_j at the latest at time $k - 1$.

All locks are returned only at the end of the transaction (by commit or abort).

Timeline for a transaction under S2PL:



S2PL prevents schedules with cascading rollbacks.

Agenda: Possible Isolation Problems and Their Solutions

Problems:

1. Dirty Read
2. Non-Repeatable Read
3. Cascading Rollback

Solution: tuple-based locking

but how exactly? short-term, long-term, 2PL, S2PL?

However, **two other problems** are still possible:

4. Phantom Problem

Solution: predicate-based locking

5. Deadlock Problem

Solution: Wait-for-graph

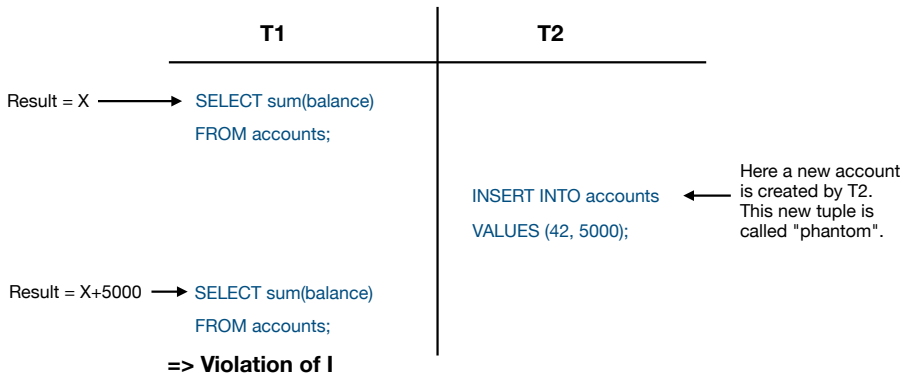
Phantom Problem

Phantom Problem

We use **Phantom Problem** to refer to the phenomenon of a transaction reading a set of tuples multiple times and potentially getting different results. This problem is similar to Non-Repeatable Read, however, the difference is:

- **Repeatable Read:** the phenomenon occurs when reading a single data object (at most one tuple, identified by its id),
- **Phantom Problem:** the phenomenon occurs when reading multiple tuples (through an arbitrary WHERE-condition).

Example: Phantom Problem

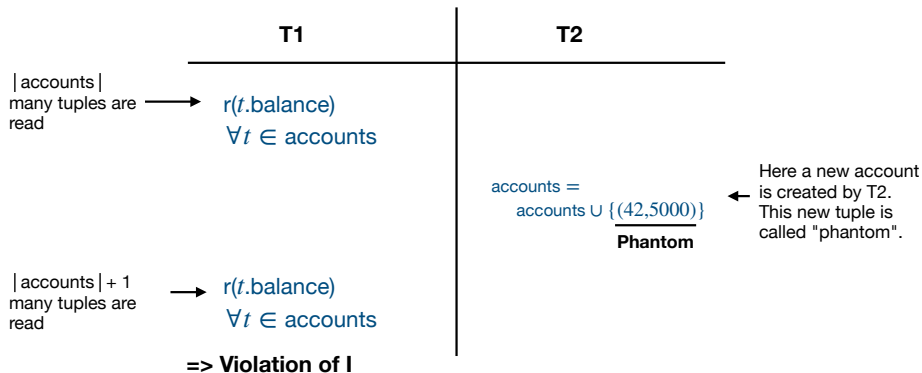


Again

What is the difference between this and the scenarios we have looked at so far?

We no longer consider only read/write operations that read or write **a concrete, guaranteed existing tuple**, but situations where via the respective WHERE clause **the set of tuples potentially changes**.

Locking Multiple Tuples

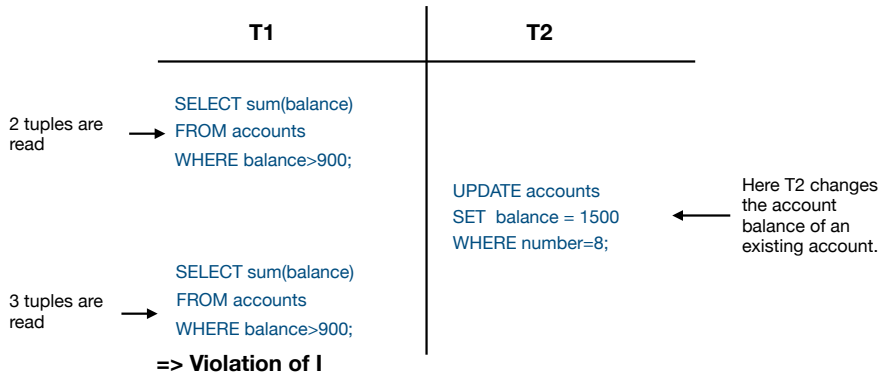


What does this mean for locking tuples?

A single statement in a transaction may request more than one read and/or write lock. In the example, $\forall t \in \text{accounts } r(t.\text{balance})$ means, that **conceptually** a separate read operation is performed for each tuple existing in the relation accounts at that time.

Phantom Problem Without Insert or Delete

Example: This schedule is permitted by S2PL!



The problem here is that different tuples are selected within the WHERE clause. The predicate `balance>900` leads to different results depending on the time of execution. Thus, isolation is violated.

Predicate Locking

Predicate Locking

To prevent a transaction with SQL statements that access multiple tuples from seeing a phantom, the corresponding value ranges must additionally be locked for such statements. We can express this through predicates.

Examples:

```
SELECT  sum(balance)
FROM    accounts;
```

actually means:

```
SELECT  sum(balance)
FROM    accounts
WHERE   True;
```

Consequences:

⇒ Lock the whole relation
'accounts' before reading!

```
SELECT  sum(balance)
FROM    accounts
WHERE   balance>900;
```

Consequences:

⇒ Before reading, lock the range
of values for which the balance is
greater than 900!

Compatibility of Predicate Locks

Notation: Requesting a predicate lock to read on predicate p by transaction T_i is noted as $getPSLock_i(p)$. $getPXLock_i(p)$ requests the analogue predicate lock for a write operation.

Compatibility of Predicate Locks

1. A new predicate read lock on relation R is allowed if there is no active predicate write lock of another transaction with predicate q for which $\{r \in D \mid p(r) = true\} \cap \{r \in D \mid q(r) = true\} \neq \emptyset$ holds.
 $D = D_1 \times \dots \times D_n$
2. A new predicate write lock on relation R is allowed if there is no active predicate lock (read or write) of another transaction with predicate q for which $\{r \in D \mid p(r) = true\} \cap \{r \in D \mid q(r) = true\} \neq \emptyset$ holds.

Example:

$getPSLock_1(R.x = 0)$ and $getPXLock_2(R.y = 42)$ are never compatible with each other, regardless of whether the tuple $(0, 42)$ is contained in the relation.

Agenda: Possible Isolation Problems and Their Solutions

Problems:

1. Dirty Read
2. Non-Repeatable Read
3. Cascading Rollback

Solution: tuple-based locking

but how exactly? short-term, long-term, 2PL, S2PL?

However, **two other problems** are still possible:

4. Phantom Problem

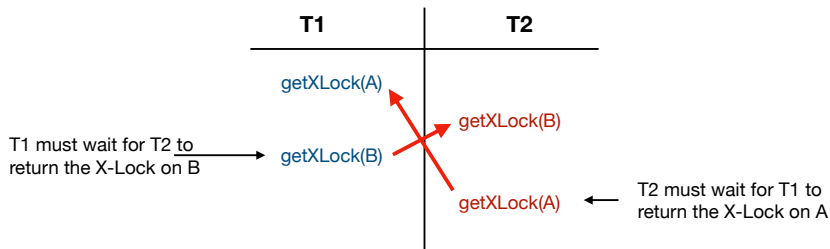
Solution: predicate-based locking

5. Deadlock Problem

Solution: Wait-for-graph

Deadlock

Example: This schedule is permitted by S2PL!



=> Deadlock of T1 and T2

Since T1 and T2 are waiting for each other, they will never continue.

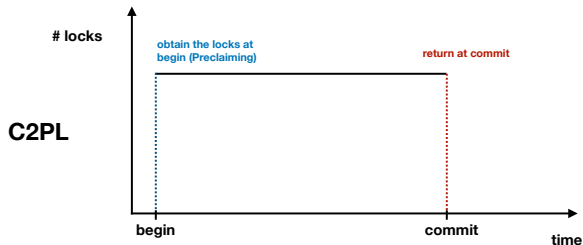
Conservative Two-Phase-Locking (C2PL, Preclaiming)

Conservative Two-Phase-Locking

All locks are requested at begin. If the transaction does not get all the locks, it does not start.

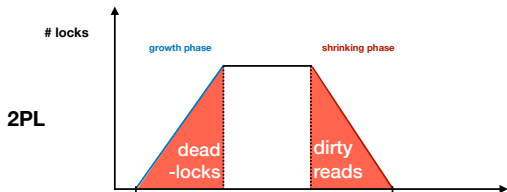
Locks are not returned before the commit.

Timeline for a transaction under C2PL:



C2PL prevents deadlocks.

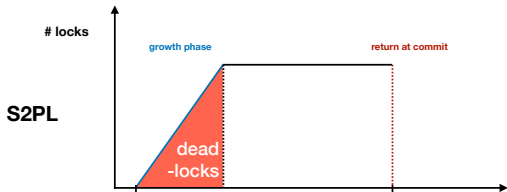
Overview of the Different 2PL Variants



deadlocks are possible

dirty reads by other transactions are possible

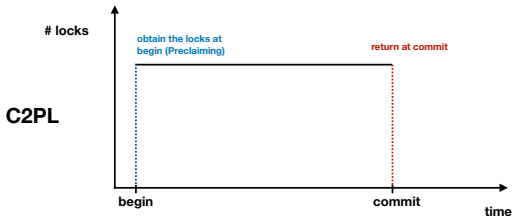
locks are requested when needed and returned early



deadlocks are possible

dirty reads by other transactions are **not** possible

locks are requested when needed and returned at commit



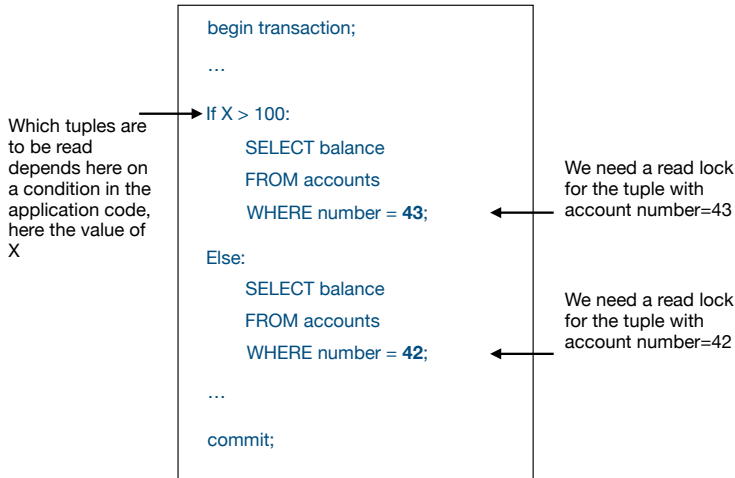
deadlocks are **not** possible

dirty reads by other transactions are **not** possible

locks are requested at begin and returned at commit

Example: Why Preclaiming Is a Bad Idea

Example code of a transaction:



Preclaiming is unrealistic and does not work for all transactions.

Wait-for-graph

Wait-for-graph

A Wait-for-graph $G = (V, E)$ consists of:

1. The set of **running** transactions $V = \{T_1, \dots, T_n\}$ as vertices, as well as
2. the set of directed edges $E = \{(T_i, T_j)\}$, i.e. a directed edge from T_i to T_j , if T_i waits for a lock held by T_j .

Do you remember the conflict graph from Part 1?

Exactly. That is basically the same thing!

Cycles in the Wait-for-graph

A cycle in the Wait-for-graph indicates a deadlock.

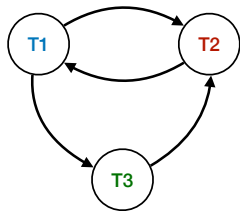
Deadlock in the Wait-for-graph

Deadlock in the Wait-for-graph

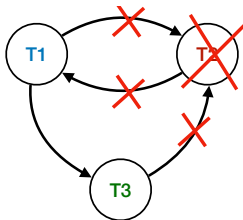
If a cycle exists in the wait-for-graph, vertices (and all edges connected to them!) must be removed until there is no cycle left, i.e. the graph is cycle-free.

Removing vertices means aborting the corresponding transactions (the application program can then start them again).

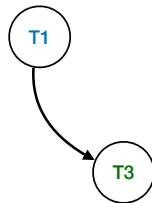
Example:



Graph with cycle



T2 abort/remove



Result:
Graph without cycle

Conflict Serialisable \neq Serializable

In the database world, there are unfortunately a confusing number of similar-sounding but different concepts.

Conflict Serialisable \nRightarrow Serializable

The schedule is conflict equivalent to a serial schedule.

The problems with that:

1. What if not all transactions in the schedule commit? This potentially violates isolation.
2. What if a single operation reads or writes more than one tuple?

That's why we need to do more:

Serializable vs Serialisable

Serializable, perfect Isolation

The isolation level Serializable is much stronger than the theoretical concept “conflict serialisable” and also prevents problems that can occur due to aborting transactions. This applies both to individual tuples and to ranges/sets of tuples that can lead to phantom problems.

Serialisable...

...can mean:

1. Synonym for ‘conflict serialisable’,
2. Synonym for Serializable, i.e. perfect isolation in the sense of the I in ACID

Isolation Levels: Weakening of Consistency Guarantees

Isolation Levels: Weakening of Consistency Guarantees

Many database systems offer the possibility to weaken the isolation of transactions against each other.

gain: higher performance due to less overhead for concurrency control

loss: weaker isolation guarantees, which (depending on the application) can lead to isolation problems with your database.

- the default setting of database systems is typically **not** perfect isolation (in database language: `SERIALIZABLE`)
- but: perfect isolation should be implemented by every database system (if necessary, this is translated internally to serial execution)
- which error situations the weaker isolation levels allow for is unfortunately highly dependent on the implementation of the concurrency component in the database system
- if in doubt, read the documentation carefully for the specific database system you are using!

Example of a 2PL-based Realisation of Isolation Levels

Isolation Level	Read	Write	Problems
Read Uncommitted	no locks	S2PL	Dirty Read, Cascading Rollback
Read Committed	short-term locks	S2PL	Non-repeatable Read
Repeatable Read	S2PL		Phantoms
Serializable (perfect isolation)	S2PL with predicate locks		none

Set Isolation Levels in SQL

```
BEGIN TRANSACTION ISOLATION LEVEL
{   SERIALIZABLE
    | REPEATABLE READ
    | READ COMMITTED
    | READ UNCOMMITTED
};
...
COMMIT;
```

This is the PostgreSQL syntax. The syntax differs depending on the database system.

In this example, the isolation level is specified for a single transaction.

Attention:

If nothing else is specified, the default isolation level in PostgreSQL is `READ COMMITTED`.

Fundamental Extremes of the Locking Procedures

Bad:

Do not lock data objects
and predicates



any number of problems
with ACID and deadlocks

Good:

other transactions never
have to wait



excellent performance

Isolation Level:

READ UNCOMMITTED

VS

Good:

very restrictively lock data
objects and predicates



no problems with ACID and
deadlocks

Bad:

other transactions often
have to wait unnecessarily



weak performance

Isolation Level:

SERIALIZABLE

Isolation Levels Simulated in Python

```
In [6]: # Execute the given schedule using the transaction manager
tx_manager.execute_schedule(schedule, dump_exec_code=False)
```

```
*****
submitted_schedule
*****
0      TX2  => BEGIN()
1      TX2  => bal2_0 = READ(table_name=accounts, rowid=0, column=Balance)
2      TX2  => ASSERT(constraint=(bal2_0 >= 100))
3      TX1  => BEGIN()
4      TX1  => bal1_0 = READ(table_name=accounts, rowid=0, column=Balance)
5      TX2  => UPDATE(table_name=accounts, rowid=0, values={'Balance': bal2_0 - 100.0})
6      TX2  => COMMIT()
7      TX1  => ASSERT(constraint=(bal1_0 >= 100))
8      TX1  => bal1_0 = READ(table_name=accounts, rowid=0, column=Balance)
9      TX1  => UPDATE(table_name=accounts, rowid=0, values={'Balance': bal1_0 - 100.0})
10     TX3  => BEGIN()
11     TX3  => bal3_3 = READ(table_name=accounts, rowid=3, column=Balance)
12     TX3  => UPDATE(table_name=accounts, rowid=3, values={'Balance': bal3_3 + 100.0})
13     TX1  => bal1_3 = READ(table_name=accounts, rowid=3, column=Balance)
14     TX3  => ABORT()
15     TX1  => UPDATE(table_name=accounts, rowid=3, values={'Balance': bal1_3 + 100.0})
16     TX1  => COMMIT()

*****
executed_schedule
*****
0      TX2  => BEGIN()
1      TX2  => bal2_0 = READ(table_name=accounts, rowid=0, column=Balance)
2      TX2  => ASSERT(constraint=(bal2_0 >= 100))
5      TX2  => UPDATE(table_name=accounts, rowid=0, values={'Balance': bal2_0 - 100.0})
6      TX2  => COMMIT()
```

see GitHub: [Transaction Manager.ipynb](#)

Trading, Banking, Ticket Systems

2. What are the data management and analysis issues behind this?

...

Question 1

How do we allow concurrent modification of data without creating erroneous data?

Concurrency Control

Question 2

How do we design this so that the process and the resulting overall system are efficient?

For example, through S2PL with predicate locks or weakened guarantees by means of isolation levels.

Modern DBMSs use multi-version concurrency control (MVCC) or a combination with S2PL (see core lecture).

Trading, Banking, Ticket Systems

4. 15 min: Transfer of the basics to the concrete application

Withdraw Money With READ COMMITTED?

T1 : gets **short-term** read lock: looks if there is enough money in the account, only if balance is greater than requested amount, money may be withdrawn; **returns** read lock!

T2 : gets **short-term** read lock: looks if there is enough money in the account, only if balance is greater than requested amount, money may be withdrawn; **returns** read lock!

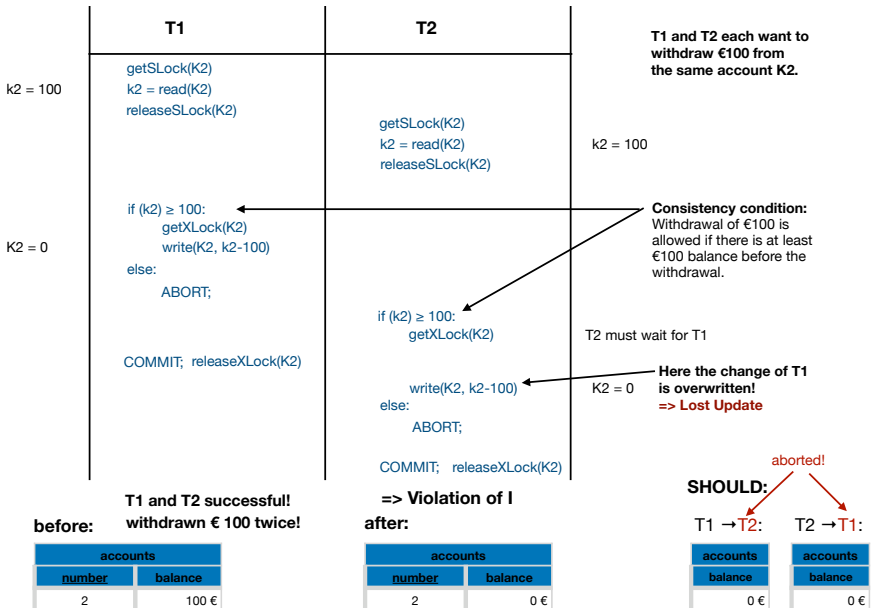
T2: gets **long-term** write lock, reduces balance and commits

T1: gets **long-term** write lock, reduces balance and commits

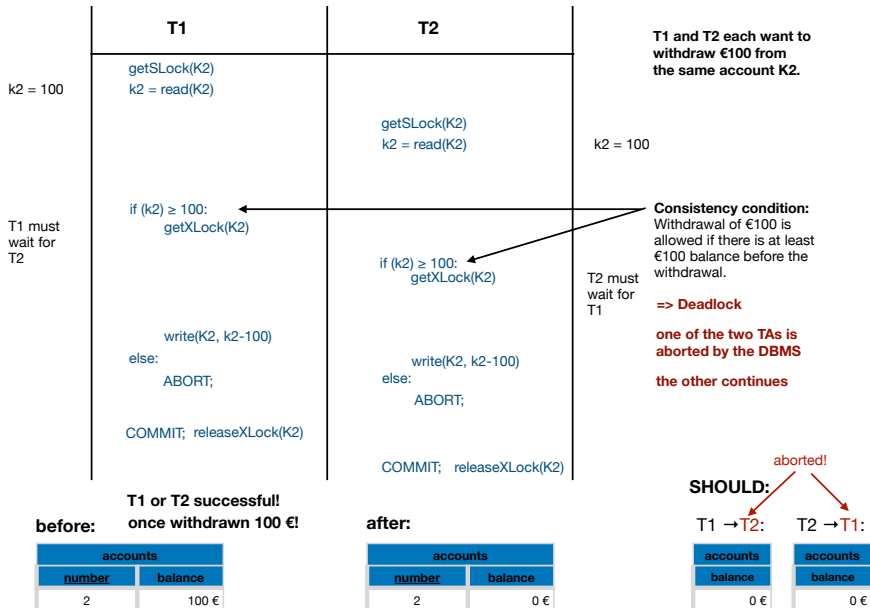
⇒ Lost change (Lost update)! However, consistency is not violated!

For a simple withdrawal scenario, READ COMMITTED is **not** sufficient.

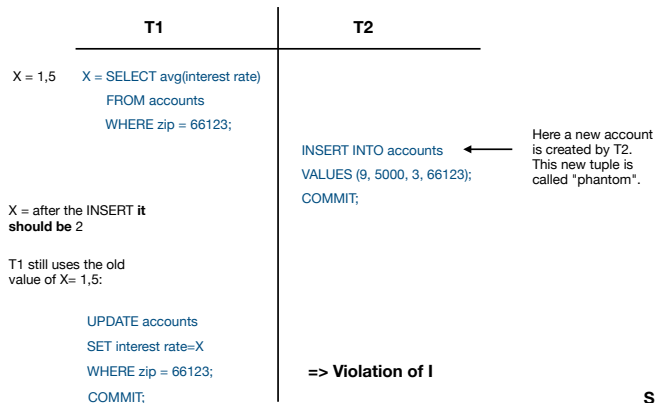
Schedule: Withdraw Money With READ COMMITTED?



Schedule: Withdraw Money With REPEATABLE READ?



Create New Account With REPEATABLE READ?



before:

accounts_new			
number	balance	interest rate	zip
2	1000 €	1	66123
1	45 €	2	66117
7	2000 €	2	66123
8	74 €	2	66117
4	500 €	1	66119

after:

accounts_new			
number	balance	interest rate	zip
2	1000 €	1,5	66123
1	45 €	2	66117
7	2000 €	1,5	66123
8	74 €	2	66117
4	500 €	1	66119
9	5000 €	1,5	66123

SHOULD:

T1 → T2:

accounts_new	
interest rate	
1,5	
2	
1,5	
2	
1	
3	

T2 → T1:

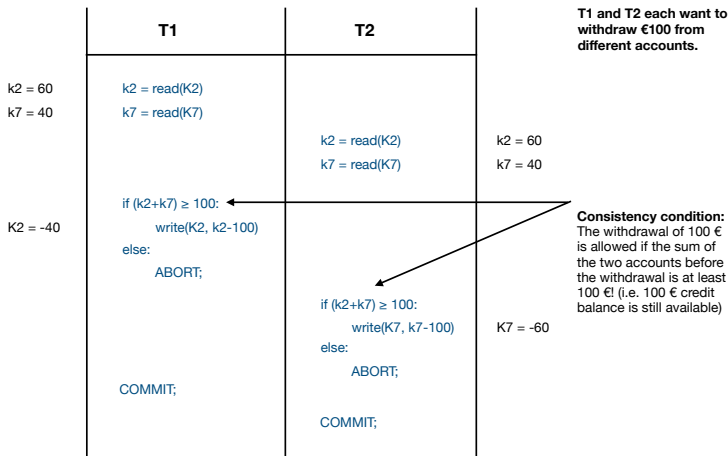
accounts_new	
interest rate	
2	
2	
2	
2	
1	
2	

Create New Account With SERIALIZABLE?

For a scenario where changes are made based **on a set of tuples** selected by WHERE, and which may give different results within a transaction, we need SERIALIZABLE. Otherwise, REPEATABLE READ is sufficient.

And now we'll take a short trip out of the university world and into reality:

Uni vs Reality: Do These Transactions Commit?



=> Violation of I and C

before:

accounts	
number	balance
2	60 €
7	40 €

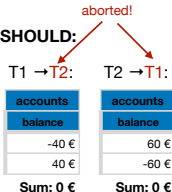
Sum: 100 €

after:

accounts	
number	balance
2	-40 €
7	-60 €

Sum: -100 €

SHOULD:



Attention: Uni

READ COMMITTED

unfortunately yes..., why?

Answer: If the read locks are only held for a short time, write locks can be issued that allow to change both accounts.

REPEATABLE READ

no, you can't, why?

Answer: Read locks are held until the end of the transaction. This means that other transactions may not be granted write locks, concurrent writing becomes impossible for T2. Moreover, a deadlock is created for this transaction.

SERIALIZABLE

no, you can't, why?

Answer: SERIALIZABLE already contains all guarantees of REPEATABLE READ...

Attention: Reality

Does this scenario run in Oracle 12c Release 2?

SERIALIZABLE

yep!

WTF?

- This problem is called *write skew*.
- It is recognised by most DBMS, e.g. PostgreSQL and MS SQL Server.
- This problem only occurs with certain classes of concurrency control algorithms: MVCC (multi-version concurrency control).
- Basic problem: both transaction work on the same version of the database (the same *snapshot*) *and* then change different accounts.
- The same account would not be a problem and would be discovered.
- Details? See core lecture *Database Systems*

Summary

Concurrency Control is Awesome

Concurrency Control allows you to fully automate concurrent access to your database without having to worry about low-level thread-synchronisation mechanisms. This is [awesome](#).

Isolation

Most database systems guarantee atomicity and isolation for transactions. Concurrent execution of transactions increases the performance of these systems enormously without creating problems.

Caution with Isolation Levels

Isolation levels are sometimes difficult to interpret. If in doubt, always use the stronger isolation level! Always check which isolation level is set by default and what this actually means in the used database system!

Cheat Sheet for Part 2

1. Numerous isolation problems possible due to reading and/or writing “dirty” or concurrently updated values.
2. Solution: tuple-based locking, either short-term, long-term, 2PL, S2PL or a mixture thereof (see isolation levels).
3. Phantom Problem: Reading via WHERE clause can potentially produce different results within a transaction.
4. Solution: predicate-based locking of areas of the relation.
5. Deadlock Problem: two or more transactions wait for each other to release their locks.
6. Solution: Wait-for-graph to be able to recognise cycles; resolving cycles through the targeted abortion of transactions.
7. Isolation levels to weaken isolation guarantees: configurable in many systems.
8. The default value of the isolation level in database systems is usually **not** **SERIALIZABLE**.

Further Material

- Chapter 9&11 in [Kemper&Eickler](#) in German
- Chapter 12&13 in Elmasri&Navathe in [German](#) or in [English](#)