Trading, Banking, Ticket Systems (Part 1)

Big Data Engineering
(formerly Informationssysteme)

Prof. Dr. Jens Dittrich

bigdata.uni-saarland.de

June 15, 2023

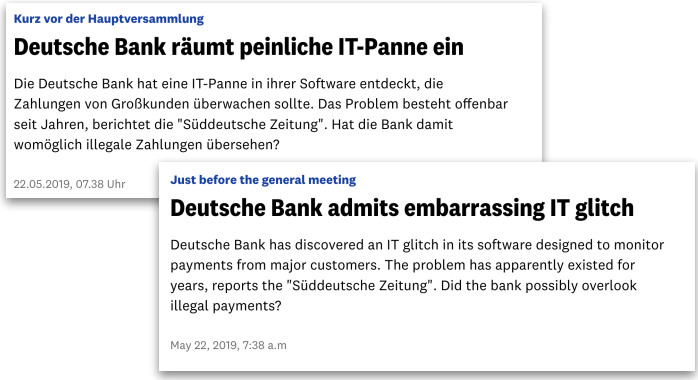
Trading, Banking, Ticket Systems

Planned structure for each two-week lecture:

1. Concrete application: Trading, Banking, Ticket Systems
2. What are the data management and analysis issues behind this?
3. Basics to be able to solve these problems
   (a) Slides
   (b) Jupyter/Python/SQL Hands-on
4. Transfer of the basics to the concrete application

Trading, Banking, Ticket Systems

1. Concrete application: Trading, Banking, Ticket Systems



Kurz vor der Hauptversammlung

Deutsche Bank räumt peinliche IT-Panne ein

Die Deutsche Bank hat eine IT-Panne in ihrer Software entdeckt, die Zahlungen von Großkunden überwachen sollte. Das Problem besteht offenbar seit Jahren, berichtet die "Süddeutsche Zeitung". Hat die Bank damit womöglich illegale Zahlungen übersehen?

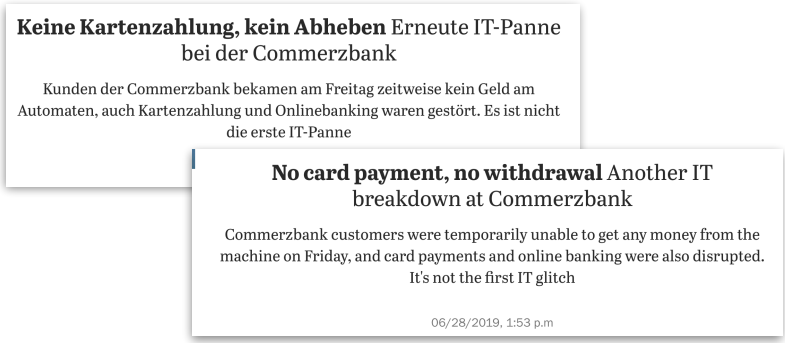
22.05.2019, 07.38 Uhr

Just before the general meeting

Deutsche Bank admits embarrassing IT glitch

Deutsche Bank has discovered an IT glitch in its software designed to monitor payments from major customers. The problem has apparently existed for years, reports the "Süddeutsche Zeitung". Did the bank possibly overlook illegal payments?

May 22, 2019, 7:38 a.m



Keine Kartenzahlung, kein Abheben Erneute IT-Panne bei der Commerzbank

Kunden der Commerzbank bekamen am Freitag zeitweise kein Geld am Automaten, auch Kartenzahlung und Onlinebanking waren gestört. Es ist nicht die erste IT-Panne

No card payment, no withdrawal Another IT breakdown at Commerzbank

Commerzbank customers were temporarily unable to get any money from the machine on Friday, and card payments and online banking were also disrupted. It's not the first IT glitch

06/28/2019, 1:53 p.m

START INS JAHR

IT-Probleme bei Bank Austria führten zu verspäteten Buchungen

Ein Programmfehler von Rechnern hatte verspätete Buchungen, auch von Pensionen und Gehältern, zur Folge. Informiert hat die Bank nicht

Renate Graber
8. Jänner 2019, 17:37

START OF THE YEAR

IT problems at Bank Austria led to late bookings

A bug in computers resulted in late postings, including pensions and salaries. The bank did not inform

Renate Graber
January 8, 2019, 5:37 p.m

Datenverlust

Gravierende IT-Panne bei der UBS – bis zu 1500 Kunden betroffen

Mi 05.12.2018 - 10:01 Uhr
von beat schmid, ch media

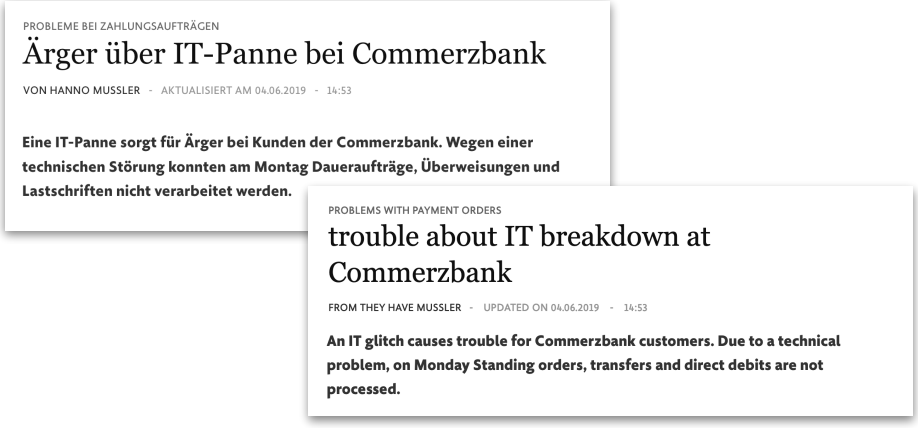
Ein IT-Fehler hat bei der UBS zu einer Datenpanne geführt. Dokumente von bis zu 1500 Kunden könnten verloren gegangen sein.

data loss

Serious IT breakdown at UBS - up to 1500 customers affected

We 05.12.2018 - 10:01 Clock
from beat schmid, ch media

An IT error has led to a data breach at UBS. Documents from up to 1500 customers could have been lost.



PROBLEME BEI ZAHLUNGSAUFTRÄGEN

Ärger über IT-Panne bei Commerzbank

VON HANNO MUSSLER - AKTUALISIERT AM 04.06.2019 - 14:53

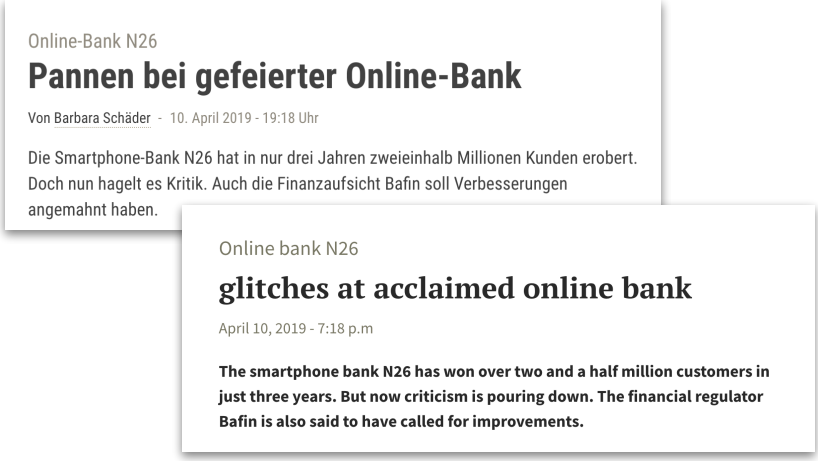
Eine IT-Panne sorgt für Ärger bei Kunden der Commerzbank. Wegen einer technischen Störung konnten am Montag Daueraufträge, Überweisungen und Lastschriften nicht verarbeitet werden.

PROBLEMS WITH PAYMENT ORDERS

trouble about IT breakdown at Commerzbank

FROM THEY HAVE MUSSLER - UPDATED ON 04.06.2019 - 14:53

An IT glitch causes trouble for Commerzbank customers. Due to a technical problem, on Monday Standing orders, transfers and direct debits are not processed.



Online-Bank N26

Pannen bei gefeierter Online-Bank

Von Barbara Schäder - 10. April 2019 - 19:18 Uhr

Die Smartphone-Bank N26 hat in nur drei Jahren zweieinhalb Millionen Kunden erobert. Doch nun hagelt es Kritik. Auch die Finanzaufsicht Bafin soll Verbesserungen angemahnt haben.

Online bank N26

glitches at acclaimed online bank

April 10, 2019 - 7:18 p.m

The smartphone bank N26 has won over two and a half million customers in just three years. But now criticism is pouring down. The financial regulator Bafin is also said to have called for improvements.

Why?

The reasons for these failures are probably very different.

But you can ask yourself a few questions as a computer scientist:

1. Why do the outages sometimes last so long?
2. Why does it sometimes take so long to get the systems back to a consistent state and resume normal operations?
3. Why are there so many failures?
4. Why are so many customers affected?
5. Given the investments, why don't these systems run more stably?

When dealing with data, there are numerous possible (and typical) sources of error. They have been known for decades.

<arrogant_mode>
Oh yes: and their solutions as well ...
</arrogant_mode>

Database Management Systems (DBMS)

Relational database systems (RDBMS, usually just DBMS) have been developed since the 1970s. Modern DBMS typically have the following features:

1. extended relational model: JSON, arrays, text, spatial data, etc.
2. very extensive SQL dialect (depending on the system)
3. **automatic** query optimiser (rule- and cost-based)
4. very high performance (mostly, depending on the system)
5. massive support for physical design (index structures, partitioning, caching, materialisation)
6. support for "modern" hardware: DRAM, PRAM, FPGAs, GPUs, ...

Database Management Systems (DBMS)

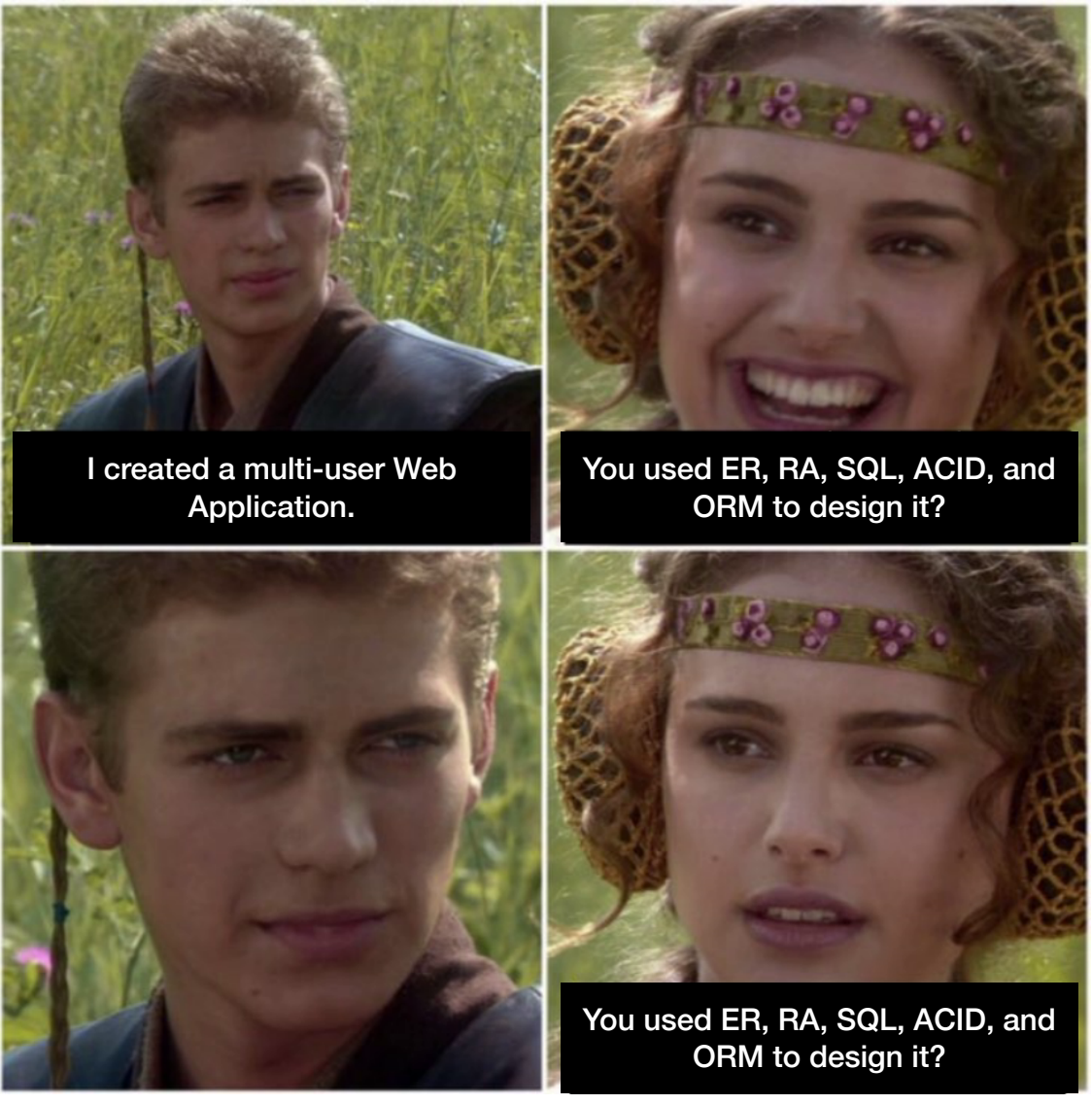
And as far as avoiding and dealing with error situations is concerned, in particular:

7. extensive support for transactions and ACID
8. concurrency control (**automatic** concurrency control) **today!**
9. **automatic** crash recovery, replication, backup
10. **automatic** consistency control (unfortunately often not used)
11. dynamic views, access rights (logical data independence)

And that is what I promised with the teaser slide when announcing this lecture:

Big Data Engineering

LS Dittrich



What you will learn in this course:

1. Foundations allowing you to build robust, scalable, and maintainable Web Applications (aka information systems)

2. How to **stop worrying** and delegate **all** the nasty, error-prone, and handwritten coding to relational database systems, including:

   A) fully-automatic transactions

   C) fully-automatic data consistency

   I) fully-automatic concurrency control

   D) fully-automatic crash recovery

   P) fully-automatic physical data independence

   Q) fully automatic query optimization

Thursdays 10:15-12:00

Stahlkocher, Thomas doerfer (CC BY-SA 3.0)

Summary: Analogy ESP (Electronic Stability Program)

before 1997

controls that prevent the problem

+ incredibly talented driver

after 1997

+ automated braking of individual wheels

(ESP)

mandatory since 2014

Important DBMS

Commercial:

- Oracle
- Microsoft
- DB2
- Actian Vector
- SAP Hana
- Exasol
- ...

Open Source:

- PostgreSQL
- MySQL
- SQLite
- mutable
- ...

Good list of (almost) all databases:
Database of Databases, https://dbdb.io/

(on call on 12.6.2023: 904 databases)

Approach in this Lecture so far

So far:

- first: in the Jupyter Notebook, load CSV files once into relations (Python, DuckDB, ...), all in a single-thread
- then: any number of read queries on the relations!

  $\Rightarrow$ no problems with concurrency!

Now:

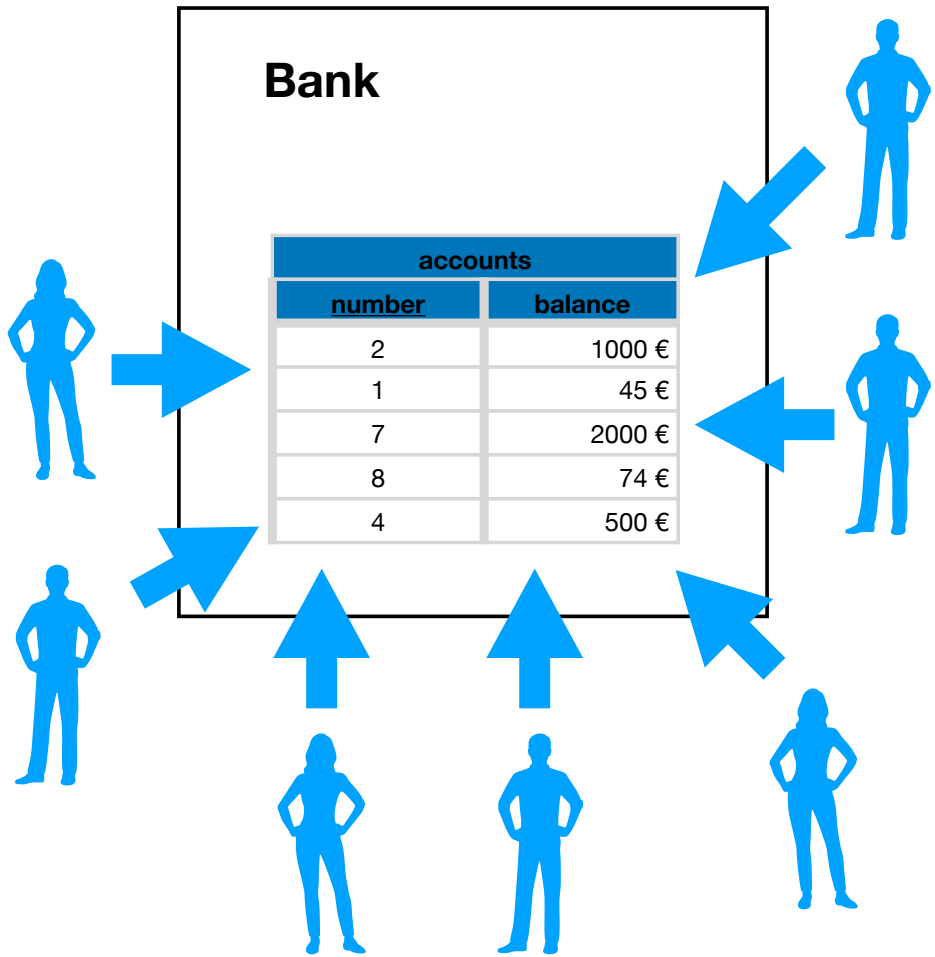
- Queries shall be allowed to change tuples in relations concurrently, *in any way they want*.

  "*in any way they want*"? Hmmm...

Problem Scenario

Problem Scenario

Multiple customers want to access their accounts at the same time, at least one customer changes his/her account balance by withdrawing, depositing, or transferring.



Let's assume one relation

[accounts] :
{[number:int, balance:int]}

Let us take a look at different variants of this problem scenario:

Withdraw and Deposit: Houston, We Have (No) Problem!

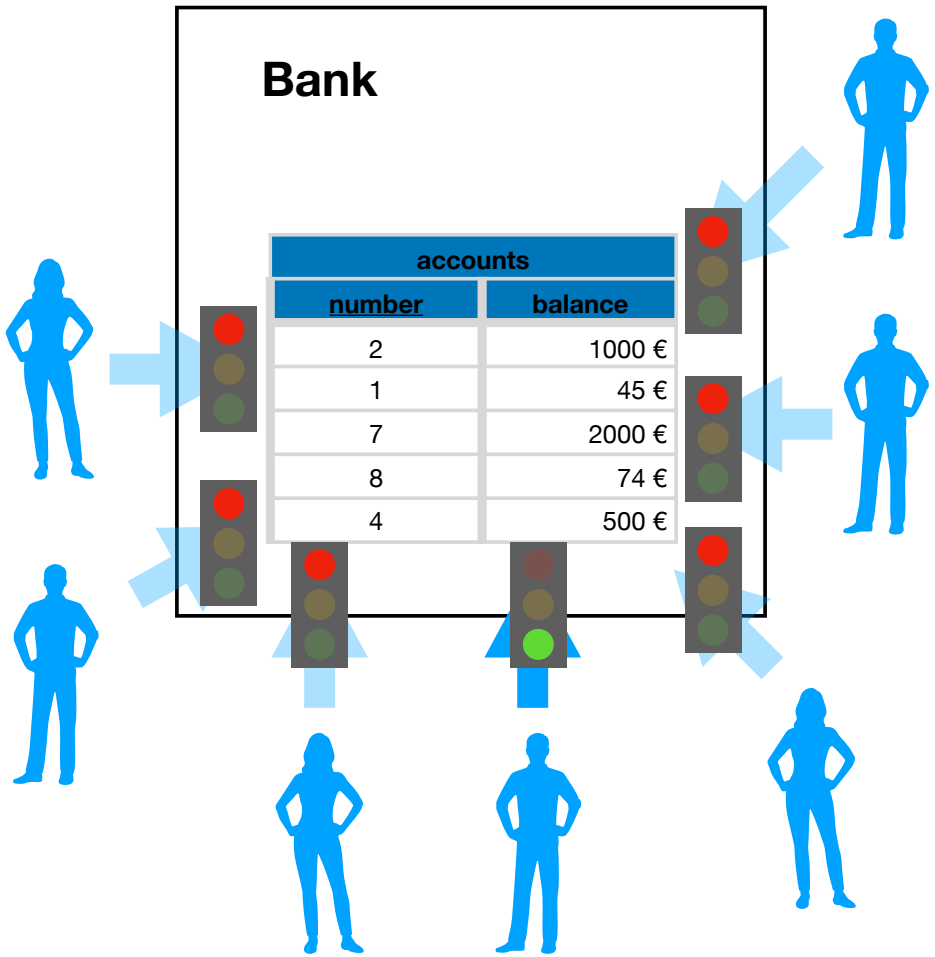
No problem if:

At any time, a maximum of one customer accesses the *accounts* relation.

Solution 1:

Block access to relation/bank for other clients

This corresponds to **a queue for all queries**: serial processing of all queries on the relation *accounts*.



The customer conceptually "owns" the entire bank for a short time!

Withdraw and Deposit: Houston, We Still Have No Problem!

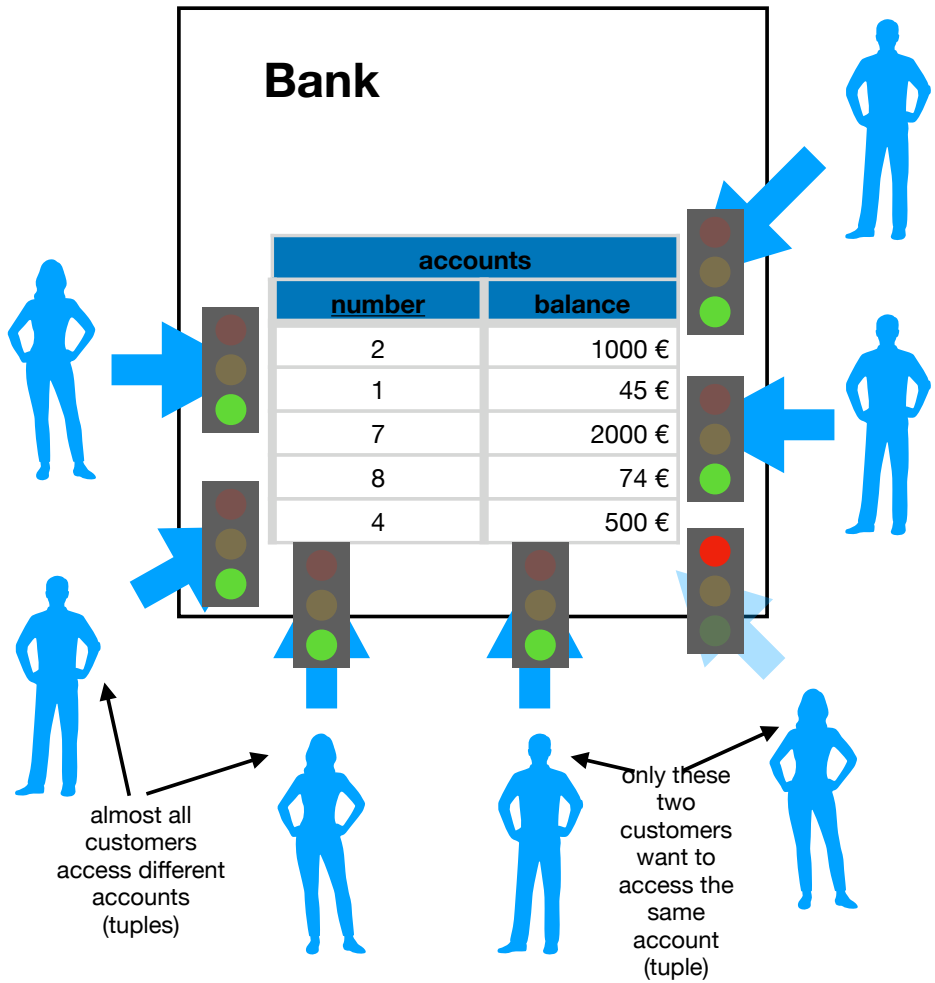
No problem if:

At any time, a maximum of one customer accesses a tuple of the relation *accounts*

Solution 2:

Block access to the respective tuple/account for other clients

This corresponds to **one queue per account**: serial processing of all queries **per tuple/account**.



The customer conceptually "owns" his/her account for a short time!

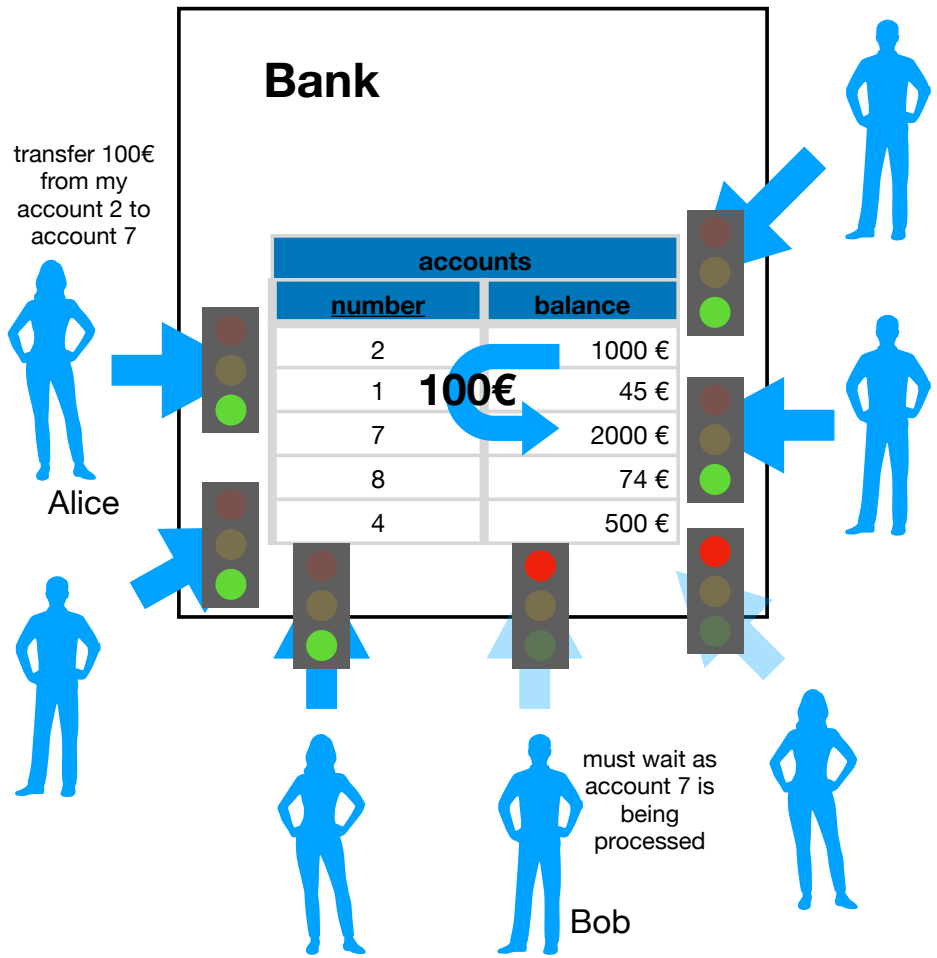
Bank transfer: Houston, Is This a Problem?

No problem if:

At any time, a maximum of one customer accesses different tuples of the relation *accounts*

Solution 3:

Block access to all affected (usually only two) tuples of the transfer for other clients



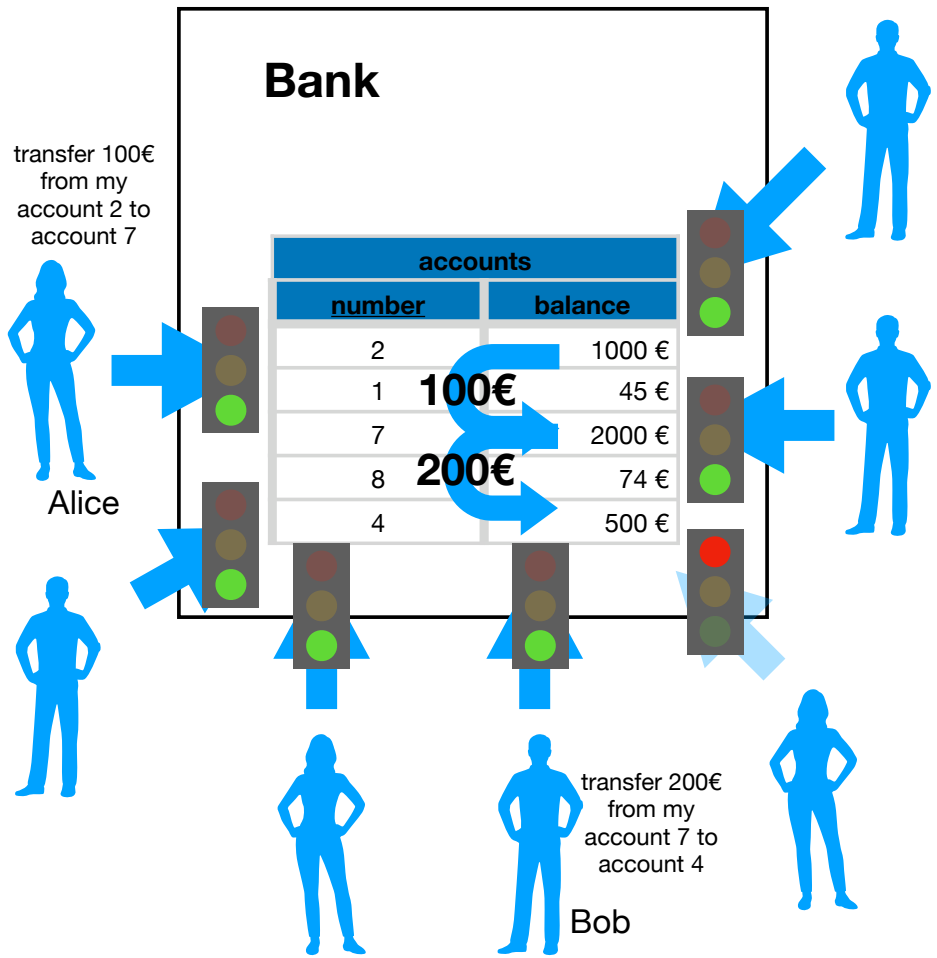
Two Transfers: Houston, We Can Handle This Alone!

No problem if:

All tuples of each transfer are respectively blocked for all others

Solution 4:

Block access to all tuples of the transfer for other customers. Implicitly, this leads to a serialisation (determination of the order) of the transfers.



This corresponds to: at all times at most one customer is served.

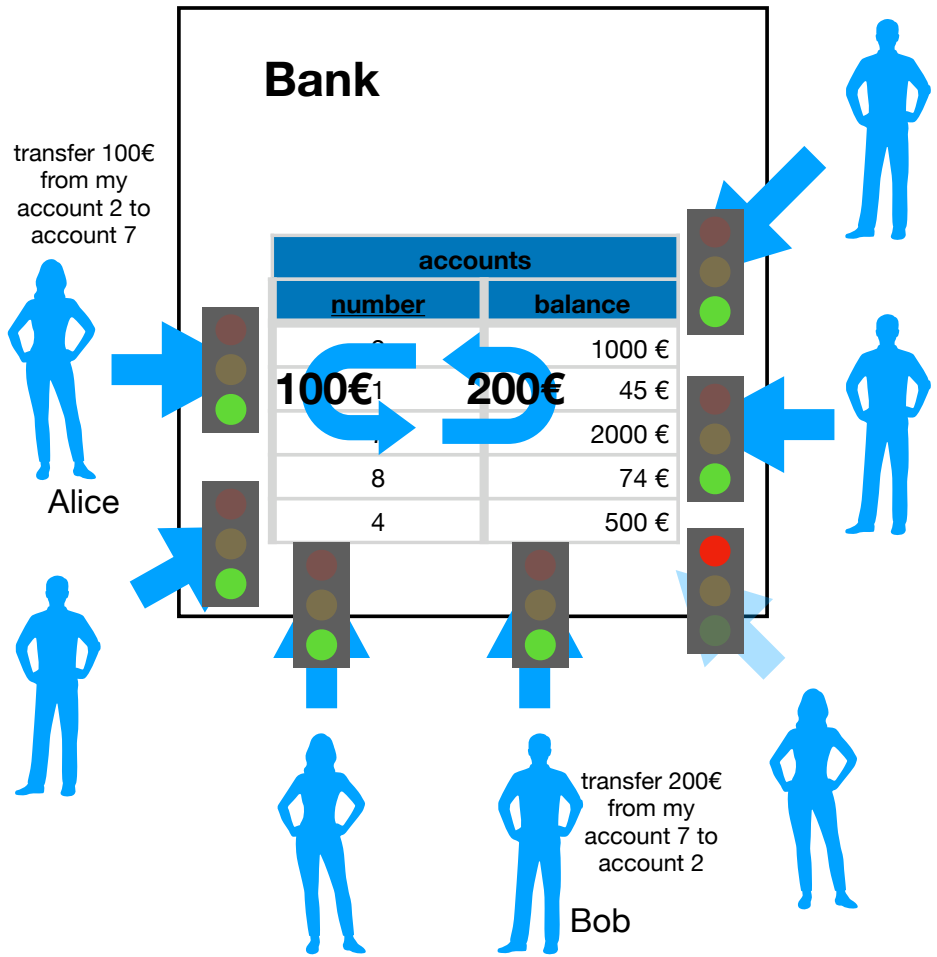
Two Transfers: Houston, We Have a Problem!

No problem if:

All tuples of each client's transfer are blocked for all others
AND: the two transfers do not jam (deadlock)

Solution 5:

???



Let's take a closer look at the details...

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?

Question 2

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

Trading, Banking, Ticket Systems

3. Basics to be able to solve these problems
   (a) Slides
   (b) Jupyter/Python/SQL Hands-on

Agenda:

database systems

transactions

ACID

serialisability theory:

- schedule
- conflict serialisability

concurrency control:

- two-phase-locking (2PL)
- isolation level
- multi-version concurrency control (MVCC) (core lecture)

Read Operation vs Write Operation

Read operation

A read operation reads **at most one tuple** from one relation.

Notation: $r(A)$: the data object A, i.e. the attribute value(s) of a tuple or at most the entire tuple, is read.

Write operation (also update operation)

A write operation changes **at most one tuple** in a relation. There are three different types of write operations:

1. **Insert**: inserts a new tuple into a relation.
2. **Update**: changes one, several or all attribute values of an existing tuple of a relation.
3. **Delete**: removes an existing tuple from a relation.

Notation: $w(A)$: the data object A, i.e. the attribute value(s) of a tuple or at most the entire tuple, is written.

Relationship to SQL

In SQL, $r(A)$ typically corresponds to:

```
SELECT  [A]
FROM    Foo
WHERE   Foo.ID=42;
```

In SQL, $w(A)$ typically corresponds to:

```
UPDATE  Foo
SET     A1 = value1, ..., AN = valueN
WHERE   Foo.ID=42;
```

or:

```
INSERT INTO Foo VALUES (value1, ..., valueN);
```

or:

```
DELETE FROM Foo
WHERE   Foo.ID=42;
```

In SELECT, UPDATE, and DELETE access takes place via the key of the relation, ID in this example.

Transaction

Transaction

A transaction T bundles one or more read and write operations into an inseparable unit. The transaction defines the *conceptual order* in which its read and write operations are to be performed.

Notation: The beginning of a transaction is marked with b (for *begin*), the successful end with c (for *commit*).
$x \to y$: "operation x takes place before operation y". Each operation must take place after the begin and before the commit (or abort).

Example:
Transaction T_1 (money transfer from account 2 to account 7, see above):
K_2: account balance of account 2, K_7: account balance of account 7

$$b_1 \to \underbrace{r_1(K_2)}_{\text{read the old value}} \to \underbrace{w_1(K_2)}_{\text{write the new value}} \to \underbrace{r_1(K_7)}_{\text{read the old value}} \to \underbrace{w_1(K_7)}_{\text{write the new value}} \to c_1$$

The Database System's View

Warning

In the last example, we **did not note anywhere** how the written value of K_2 and K_7 was changed in the programming language! Why?

Host programming language statements vs. read/write operations in SQL

Software mixes statements of the host programming language (Java, C , Python, etc.) with read and write operations (SQL). The system that processes these SQL queries does not necessarily see the statements of the programming language.

- Therefore, we assume in the following that we only see the read and write operations (the SQL statements), i.e. the same information the database system sees.
- In other words, with this information alone, we must be able to decide whether certain reading and writing operations could be problematic or not.

Transaction: Formal

Transaction

A transaction is a sequence of read and write operations.
$T_i = b_i \rightarrow o_i^1(A_1) \rightarrow \ldots \rightarrow o_i^n(A_n) \rightarrow f_i$, where $f_i \in \{a_i, c_i\}$ indicates, whether T_i aborts or commits, $o_i \in \{r_i, w_i\}$ for all $o_i^1(A_1), \ldots, o_i^n(A_n)$ being the read and write operations of the transaction and $A_1, \ldots, A_n$ the data objects, on which T_i works. If we do not need the numbering of the operations or the beginning of the transaction is negligible, we omit it. We denote the order relation that orders the operations of T_i according to the sequence with $ord(T_i)$. $ord(T_i)$ is a strict total order.

Example:

Transaction T_1 (money transfer from account 2 to account 7, see above):

$b_1 \rightarrow r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow w_1(K_7) \rightarrow c_1$

$$ord(T_1) = \Big\{\big(b_1, r_1(K_2)\big), \big(r_1(K_2), w_1(K_2)\big), \underbrace{\big(b_1, w_1(K_2)\big)}_{\text{redundant because of transitivity}} \ldots\Big\}$$

State of the Instance, Sequential, and Concurrent Execution

State of the instance after execution of a transaction

Let T_i be a transaction that is executed on a relation R and commits (analogous notation for multiple relations). Then we denote the state of the instance **after** executing the transaction with $\{R\}_{T_i}$.

Sequential execution

We denote the **sequential execution** of two transactions T_1, T_2 on R with $\{R\}_{T_1|T_2}$, which means that T_1 is executed **before** T_2.

Concurrent execution

We denote the **concurrent execution** of two transactions T_1, T_2 on R with $\{R\}_{T_1\|T_2}$.

Atomicity, A

Atomicity, A

A transaction is an inseparable unit. The read and write operations of a transaction are either executed completely or not at all. Therefore, the state of the database R after the execution of T is either $\{R\}$ or $\{R\}_T$.

Examples:
fully executed transaction by Alice (transfer 100€ from account 2 to account 7, **committed**):

$b_1 \rightarrow r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow w_1(K_7) \rightarrow c_1$
Here all actions were executed.

partially executed transaction (**aborted**):

$b_1 \rightarrow r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow a_1$

Here only $w_1(K_2)$ was executed but not $w_1(K_7)$. We need to ensure that the partial changes made by $w_1(K_2)$ are not applied to the database.

Atomicity Implies:

No halfhearted ~~things~~ transactions

The database (the set of relations and the data stored in those relations) must be protected against only parts of changes of a transaction being executed on the database!

Difference to Isolation

Atomicity describes the behavior **within** an individual transaction, whereas Isolation describes the behavior **among** one or more transactions.

Consistency, C

Consistency Condition

A consistency condition $cp : [R] \rightarrow bool$ is an invariant to the database. It sets a condition on the database.

Consistency, C

A transaction T leaves the database in a consistent state when committing: $cp(\{R\}_T) = true$.
During the processing of transaction T, the database may be arbitrarily inconsistent (**from the point of view of T!**).

Example: Consistency condition: for transfers within a bank, *the sum over all account balances is always the same*.

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

This returns **always** the same result, **before and after** execution of a transfer within the bank. But not necessarily, **while** performing the transfer (from the point of view of the transaction doing the transfer).

Consistency Conditions

Examples:

- Key:
  `PRIMARY KEY`
- Foreign key exists:
  `FOREIGN KEY (Person_ID) REFERENCES Persons(ID)`
- No namesakes:
  CONSTRAINT names UNIQUE (firstName, lastName)
- Prof holds a maximum of three lectures per semester:
  CREATE TRIGGER ...
  Details here https://youtu.be/aTeRR9XmPWE (in German)
  Rules vs Trigger in PostgreSQL:
  https://www.postgresql.org/docs/current/rules-triggers.html

Consistency Implies:

Check consistency conditions for each transaction!

The database (the set of relations and the data stored in those relations) must be protected against changes of a transaction being applied to the database that violate any consistency condition. In other words, before a transaction is allowed to commit, the database system must ensure that the changes done by that transaction do not violate any of these consistency conditions.

Isolation, I

Isolation, I

Concurrent transactions do not affect each other. For each transaction T_i, it appears like T_i is the only transaction that is executed. What other transactions $T_{j \neq i}$ do or their effects on the database are not visible to T_i.

A sufficient condition for this is that the concurrent execution of two transactions T_1, T_2 transitions the database into a state that corresponds to **one** sequential execution of both transactions:

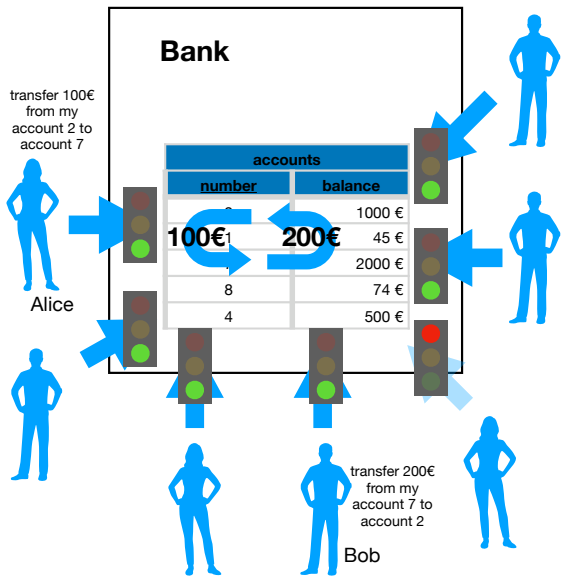
$$\{R\}_{T_1 \| T_2} == \{R\}_{T_1 | T_2} \textbf{ or } \{R\}_{T_1 \| T_2} == \{R\}_{T_2 | T_1}$$

and that the read operations of the transactions return results that correspond to the same sequential execution.

Isolation, I

Example:

However these money transfers are executed: **one after the other** (in whatever order) or **concurrently** (the individual read and write operations of the transfers are somehow interlocked):



The world view of each transaction is: I am the only transaction that is currently being executed.

Isolation Implies:

Do not leave concurrency to chance!

The database (the set of relations and the data stored in those relations) must be protected against concurrently executed transactions violating properties I or C. We must protect the database from such changes.

We will look at that in detail this week and next week.

Durability, D

Durability, D

If a transaction T commits, the effects of its write operations (i.e. the effects of change operations applied by it) must be preserved in the database, no matter what.

Example:

1. Transfer transaction commits,
2. then there is a power failure.

Were the changes persisted on hard disk/SSD before the power failure? And if so, which ones? And how do we determine this?

Is it possible that only parts of the changes were persisted?
$\Rightarrow$ additional problem with atomicity (A) and consistency (C)?

Hmmmm....

Durability Implies:

We need to protect the database against errors of any kind

Algorithms and systems must be designed in such a way that D is guaranteed in the event of errors of any kind (from software errors to catastrophic errors).

That is not that easy. But it can be done.

And what is a "catastrophic error" anyway?

We look at this topic in detail in the core lecture "Database Systems".

Serialisability Theory: Schedule (History)

Schedule

A schedule S specifies for a set of transactions $T_1, \ldots, T_n$ the order of their read, write, and commit or abort operations.

We write down a schedule as a strict partial order over the individual operations of $T_1, \ldots, T_n$. Note that the order among the operations inside a transaction is preserved: $ord(T_i) \subset S$ for all $1 \leq i \leq n$.

Notation:
We do not explicitly mark the beginning of a transaction T_i in the schedule (above we had noted this for individual transactions with b_i): in a schedule, the first operation of a transaction implicitly marks the start of that transaction.

Conflict Operations

Conflict Operations

Two read/write operations $o_i(A), p_j(B)$ with $o = r \vee o = w$ and $p = r \vee p = w$ of a schedule are called conflict operations if **all** of the following conditions apply:

1. they belong to different transactions $i \neq j$,
2. both access the same data object $A = B$,
3. at least one of them is a write operation $o = w \vee p = w$.

Examples:

$r_1(A)$ and $r_2(A)$:	no conflict operations, only read operations
$r_1(A)$ and $w_1(A)$:	no conflict operations, same transaction
$r_1(A)$ and $w_2(A)$:	conflict operations, **all** conditions met
$w_1(A)$ and $w_2(A)$:	conflict operations, **all** conditions met
$w_1(A)$ and $w_2(B)$:	no conflict operations, not the same data object

Schedule With Total Order

Schedule With Total Order

A total-order schedule specifies the order for each pair of read, write and commit or abort operations in schedule S.

Example:

Transaction from Alice (transfer 100€ from account 2 to account 7):
$r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow w_1(K_7) \rightarrow c_1$

Transaction from Bob (transfer 100€ from account 7 to account 2):
$r_2(K_7) \rightarrow w_2(K_7) \rightarrow r_2(K_2) \rightarrow w_2(K_2) \rightarrow c_2$

A possible schedule with **total order** for this:
$r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow r_2(K_7) \rightarrow w_1(K_7) \rightarrow c_1 \rightarrow w_2(K_7) \rightarrow r_2(K_2) \rightarrow w_2(K_2) \rightarrow c_2$

How do we decide if this schedule is a good idea?

Schedule With Partial Order

Schedule With Partial Order

A schedule with partial order does not force the operations into a total order, but specifies **at least** the order for each pair of **conflict operations** in schedule S.

Example:

$$\begin{array}{llll} r_1(A) \rightarrow & w_1(A) \rightarrow & w_1(B) \rightarrow & c_1 \\ & & \uparrow & \\ & r_2(B) \rightarrow & w_2(B) \rightarrow & r_2(C) \rightarrow \quad c_2 \end{array}$$

In this schedule, the arrow $r_2(B) \rightarrow w_1(B)$ was omitted (which, according to our definition, should actually be specified).

But it holds $r_2(B) \rightarrow w_2(B)$ and $w_2(B) \rightarrow w_1(B)$

$\Rightarrow r_2(B) \rightarrow w_1(B)$.

Conflict Order

Conflict Order

The conflict order $DEP(S)$ of a schedule S is the subset of S that orders only the operations of the conflict operations.

$$DEP(S) = \{o_i \to o_j \mid o_i, o_j \in S \wedge o_i, o_j \text{ are conflict operations}\}$$

Example:

$$\begin{array}{llll} r_1(A) \to & w_1(A) \to & w_1(B) \to & c_1 \\ & & \uparrow & \\ & r_2(B) \to & w_2(B) \to & r_2(C) \to \quad c_2 \end{array}$$

$DEP(S) = \{r_2(B) \to w_1(B), w_2(B) \to w_1(B)\}$
In this schedule, the arrow $r_2(B) \to w_1(B)$ was omitted (which, according to our definition, should actually be specified).

Serial

Serial

A schedule S is called **serial**, if all transactions in S are executed completely one after the other. In other words, at any given time, only one transaction is active and fully executes all its operations.
Formal: there is a permutation π, such that $S = T_{\pi(1)} \rightarrow \ldots \rightarrow T_{\pi(n)}$

Number of serial schedules

For n transactions there are $n!$ possible serial schedules.

Examples:

$r_1(A) \rightarrow w_1(A) \rightarrow c_1 \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2$
is serial

$r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow r_1(A) \rightarrow w_1(A) \rightarrow c_1$
is serial

$r_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$
is **not** serial

Conflict Equivalent

Conflict Equivalent

Two schedules S1 and S2 are called **conflict equivalent** if all operations of conflict operations in S1 and S2 are specified in the same order: $DEP(S1) = DEP(S2)$. **Notation:** S1 $\equiv$ S2

Examples:

S1: $r_1(A) \rightarrow r_2(B) \rightarrow w_1(A) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$

S2: $r_1(A) \rightarrow w_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$

$\Rightarrow$ S1 $\equiv$ S2

S3: $r_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$

The operations in conflict operations $w_1(A)$ and $r_2(A)$ were exchanged!

$\Rightarrow$ S1 $\not\equiv$ S3 and S2 $\not\equiv$ S3.

Attention

The relative order of operations within a transaction is (still) not changed!

Conflict Serialisable

Conflict Serialisable

A schedule S is called **conflict serialisable** if it is conflict equivalent to a serial schedule.

Swapping non-conflict operations

If any schedule S can be transformed into a serial schedule by **swapping non-conflict operations**, S is conflict serialisable.

Examples:

S: $r_1(A) \rightarrow r_2(B) \rightarrow w_1(A) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$

can be converted into by swapping:
$r_1(A) \rightarrow w_1(A) \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2 \rightarrow c_1$
and then:
$r_1(A) \rightarrow w_1(A) \rightarrow c_1 \rightarrow r_2(B) \rightarrow r_2(A) \rightarrow w_2(B) \rightarrow c_2$

Since this is a serial schedule $\Rightarrow$ S is conflict serialisable.

Conflict Graph (Precedence Graph)

Conflict Graph (Precedence Graph)

The conflict graph (precedence graph) $G = (V, E)$ of a schedule S consists of

1. The set of transactions $V = \{T_1, \ldots, T_n\}$ (vertices), and
2. an edge for each pair of transactions from S that have a pair of conflict operations $E = \{(T_i, T_j) \mid (o_i, o_j) \in DEP(S)\}$.

The reading of the directed edge between transactions in the conflict graph **is an aggregated view of the conflict operations** in the corresponding schedule (... `GROUP BY transactions`...).

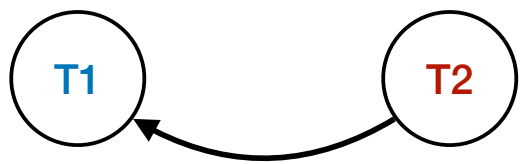
Example: (For our Alice and Bob scenario from above)

From the schedule...

$r_1(K_2) \rightarrow w_1(K_2) \rightarrow r_1(K_7) \rightarrow w_1(K_7) \rightarrow c_1$

$r_2(K_7) \rightarrow w_2(K_7) \rightarrow r_2(K_2) \rightarrow w_2(K_2) \rightarrow c_2$

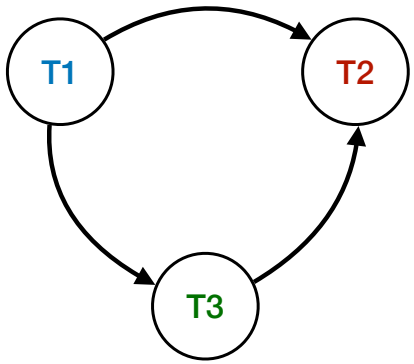
...we can directly infer the conflict graph:



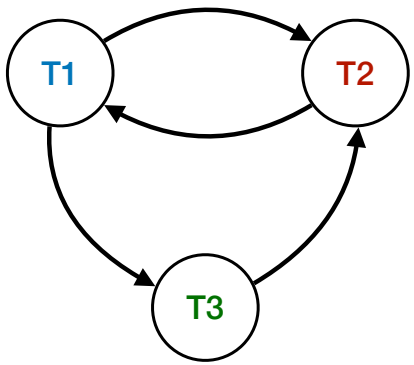
Examples

Here we have omitted the commits, as they play no role in the conflict graph.

S1: $r_1(A) \rightarrow r_3(B) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow r_3(A) \rightarrow w_2(A)$



S2: $r_1(A) \rightarrow r_3(B) \rightarrow w_1(A) \rightarrow w_2(B) \rightarrow r_3(A) \rightarrow r_1(B) \rightarrow w_2(A)$



Why does the additional edge appear in the conflict graph?

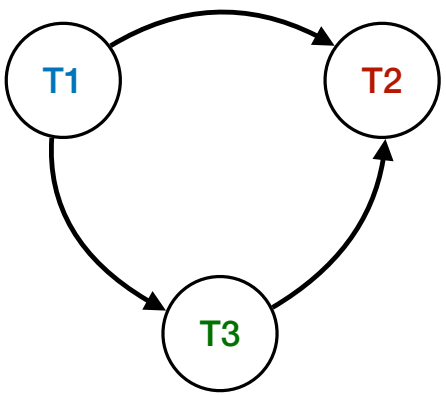
Because in the schedule S2 there are the conflict operations $w_2(B)$ and $r_1(B)$.

Cycle

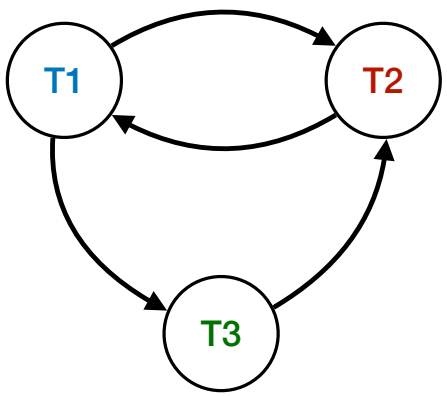
Cycle

With cycle we denote each path in the conflict graph $T_i \rightarrow T_{j \neq i} \rightarrow \ldots \rightarrow T_i$. In other words, if we start the traversal of the graph at T_i, the node T_i is reached by the traversal (here we traverse at least one other node). The graph is thus cyclic and **not a DAG** (directed acyclic graph).

Example:



cycle-free graph (DAG)



graph with cycle (not a DAG)

Conflict Serialisability in the Conflict Graph

Conflict Serialisability in the Conflict Graph

A schedule is conflict serialisable if and only if the associated conflict graph is cycle-free.

From this follows directly an important (and practicable) algorithm to make a non-cycle-free conflict graph conflict-serialisable:

Creating a cycle-free conflict graph

A non-cycle-free conflict graph can be transformed into a cycle-free conflict graph by removing all transactions that lead to cycles.

Batch vs Operation-at-a-time

Batch vs Operation-at-a-time

1. **Batch**: A set of transactions is to be executed. We can perform the above graph analysis to generate a serialisable schedule for this. For instance, using topological sorting.
2. **Operation-at-a-time**: A set of transactions sends operations to the database system in arbitrary order. How can we ensure that these concurrent operations behave like a serialisable schedule — and do so **without waiting for all transactions to send their commit!**?

Example of Batch in:

[Ankur Sharma, Felix Martin Schuhknecht, Divya Agrawal, Jens Dittrich: Blurring the Lines between Blockchains and Database Systems: the Case of Hyperledger Fabric. SIGMOD 2019.]

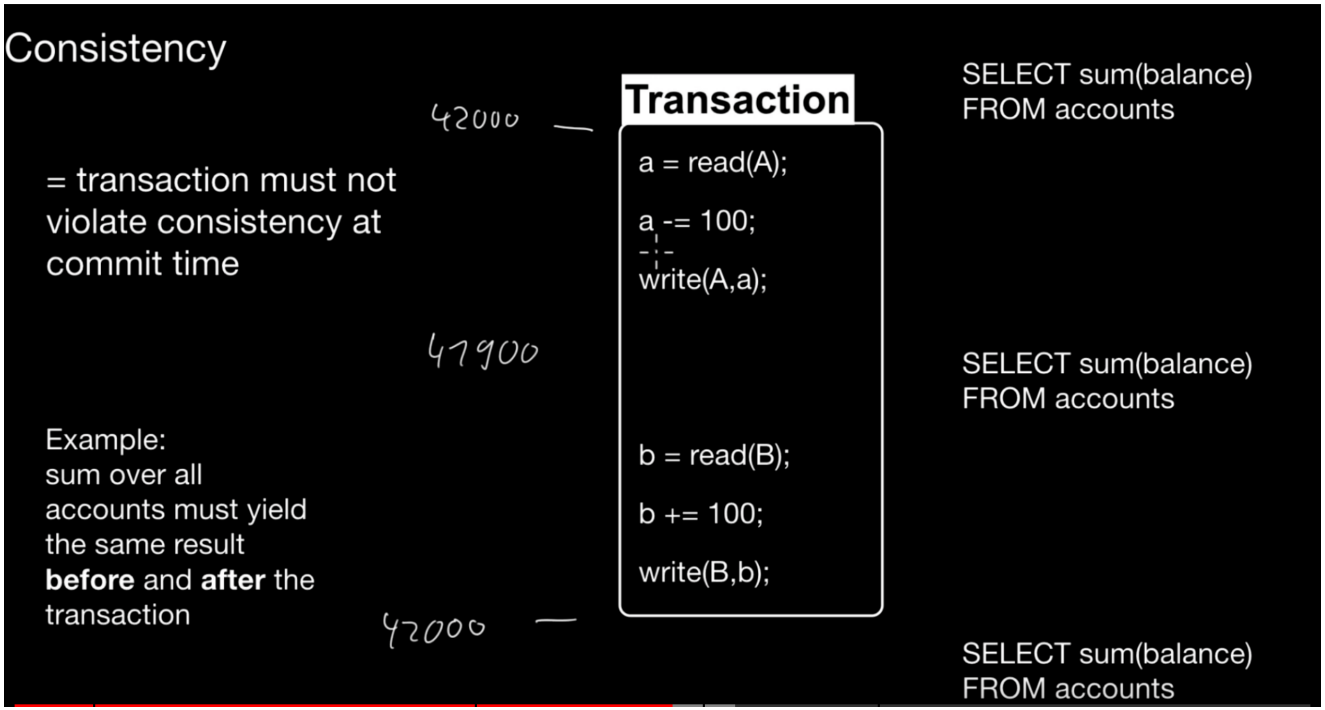
Operation-at-a-time Algorithms

Operation-at-a-time Algorithms

Most often, transaction processing assumes *operation-at-a-time*. For this we need an algorithm that guarantees that concurrent transactions behave like a serialisable schedule.

- There are numerous *operation-at-a-time* algorithms.
- In the following we show the (historically) most important one: *2PL (two-phase locking)*.
- However, modern DBMSs use *MVCC (multi-version concurrency control)*.
- MVCC can be combined with 2PL, see core lecture

Further Material



- Video in German "Transaktionen und ACID"
- Video in English "Transactions and ACID"
- Chapter 9&11 in Kemper&Eickler in German
- Chapter 12&13 in Elmasri&Navathe in German or in English