Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Databases and Information Systems (DIS)

Dr. Annett Ungethüm

Universität Hamburg

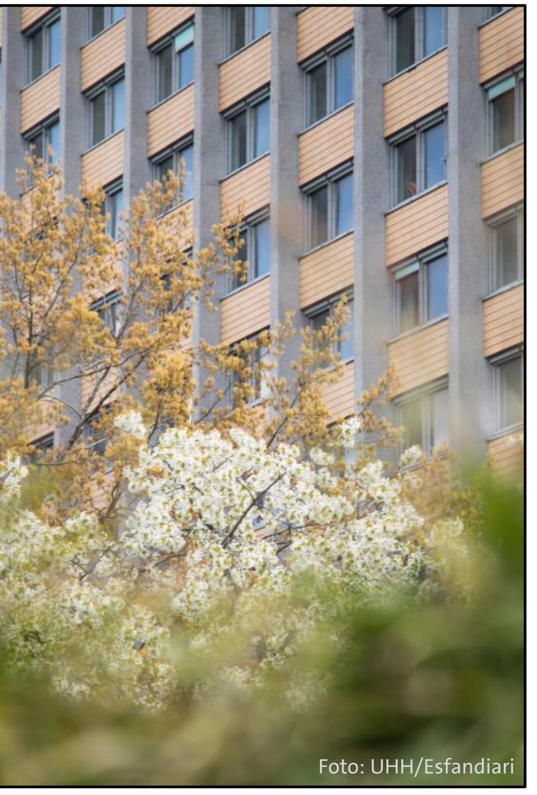
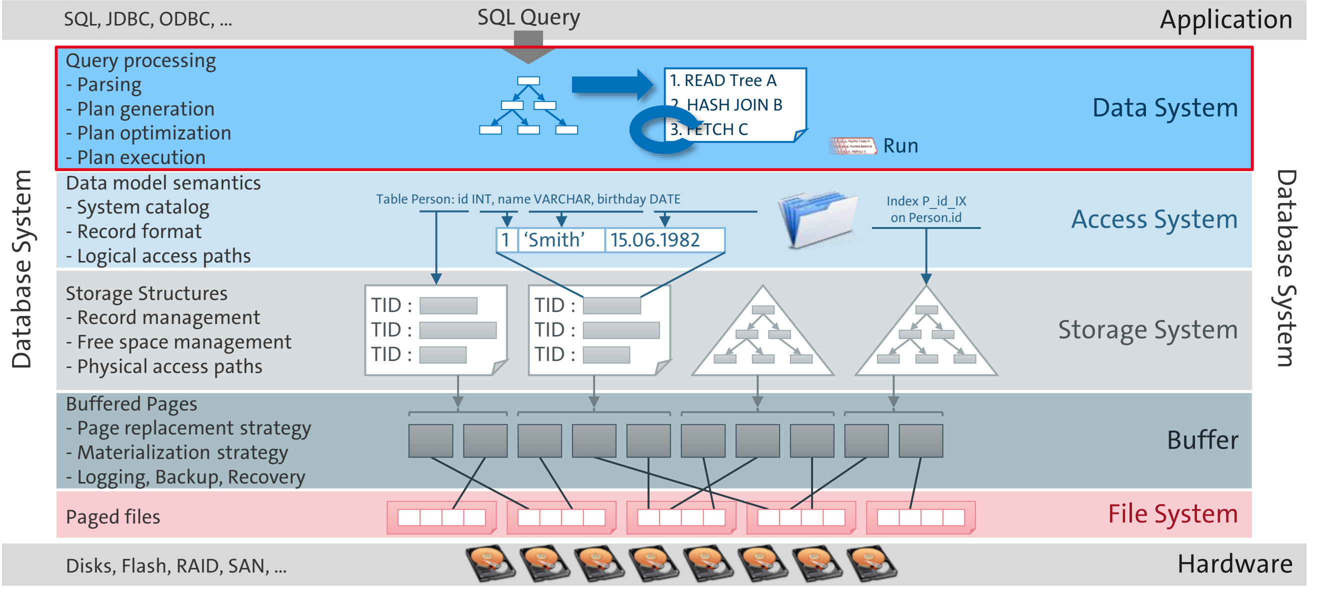


Foto: UHH/Esfandiari

The 5 Layer Model



Logical Query Execution Plan

MensaMeals

Meal	Price
Pizza	6,50
Pasta	4,90
Pie	1,20
Potato Salad	5,80
Pannfisch	7,90

DailyOffers

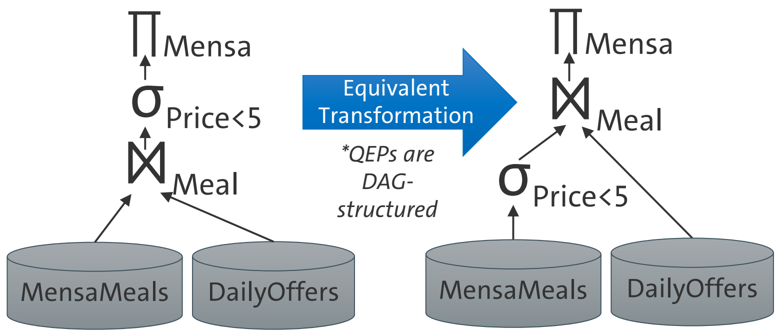
Mensa	Meal
Campus Mensa	Pizza
Mensa Cafe	Pie
Garden Mensa	Pasta
Old Mensa	Potato Salad

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

SELECT Mensa FROM MensaMeals, DailyOffers
WHERE MensaMeals.Meal = DailyOffers.Meal
AND MensaMeals.Price < 5;

Plan A: $\prod_{Mensa}(\sigma_{Price<5}(MensaMeals \bowtie_{Meal} DailyOffers))$

Plan B: $\prod_{Mensa}((\sigma_{Price<5}(MensaMeals)) \bowtie_{Meal} DailyOffers)$

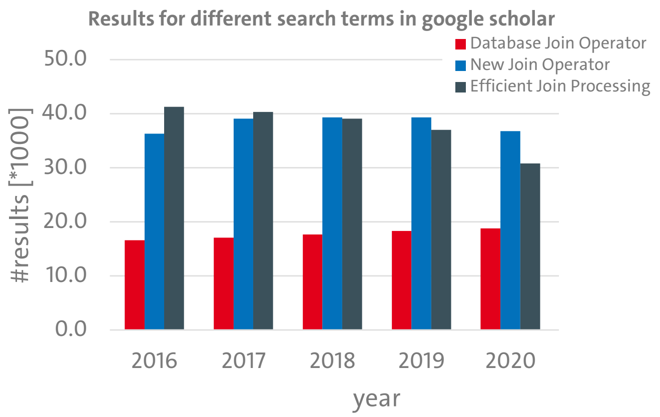


- Database Systems use a relational algebra for internal representation (see database lecture of your bachelor program)
- Optimizers try to automatically find the most efficient sequence of operators
  - ➔ Conventional approach: Reduce data as early and as cheap as possible
  - ➔ Tool: Cardinality/Selectivity estimation
- The chosen sequence of operators is the final Query Execution Plan (QEP)

Further Reading
Foundations for operator order optimization: *Bringing Order to Query Optimization*, Slivinskas et al.
https://www.researchgate.net/publication/2916321_Bringing_Order_to_Query_Optimization
Survey on different cardinality estimation techniques: *Cardinality estimation: An Experimental Survey*, Harmouch and Naumann
http://www.vldb.org/pvldb/vol11/p499-harmouch.pdf

Physical Operator Selection

- For each **logical operator** (e.g. join), there can be different **physical operators** (e.g. hash-join, nested-loop-join), i.e. the same operator can be implemented in different ways



Numbers from April 2021

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

4

- Joins are a bottleneck in most queries → Join optimization is a much-noticed field of research
- Choice of physical operator depends on exact use case. Examples from PostgreSQL:

➔ Nested-Loop: full join, one very small table, condition is not an equality
➔ Hash Join: similarity joins, small expected hash table
➔ Merge Join: sorted data, large tables

➔ EXPLAIN can be used to see the generated execution plan, often including the physical operators

Further Reading

More on join order optimization: *Query optimization through the looking glass, and what we found running the Join Order Benchmark*, V.Leis et al.

Overview on Popular Join algorithms and an alternative:
New algorithms for join and grouping operations, G. Graefe

Output of EXPLAIN

Some show a graph (duckdb)...

```
explain_key     explain_value
VARCHAR VARCHAR
[ Rows: 1]
physical_plan
SIMPLE_AGGREGATE
count_star()

PROJECTION
42

HASH_GROUP_BY
#0

PROJECTION
AUTHOR

HASH_JOIN
INNER
IDCODE=PDB_ID

SEQ_SCAN          SEQ_SCAN
authors           testsequence
IDCODE            Sequence_Count
AUTHOR            PDB_ID
                  Filters:
                  Sequence_Count<3
```

...some show an ugly graph (sqlite)...

```
QUERY PLAN
|--CO-ROUTINE 1
|  |--SCAN TABLE testsequence
|  |--SEARCH TABLE authors USING AUTOMATIC COVERING INDEX (IDCODE=?)
|  `--USE TEMP B-TREE FOR GROUP BY
`--SCAN SUBQUERY 1
```

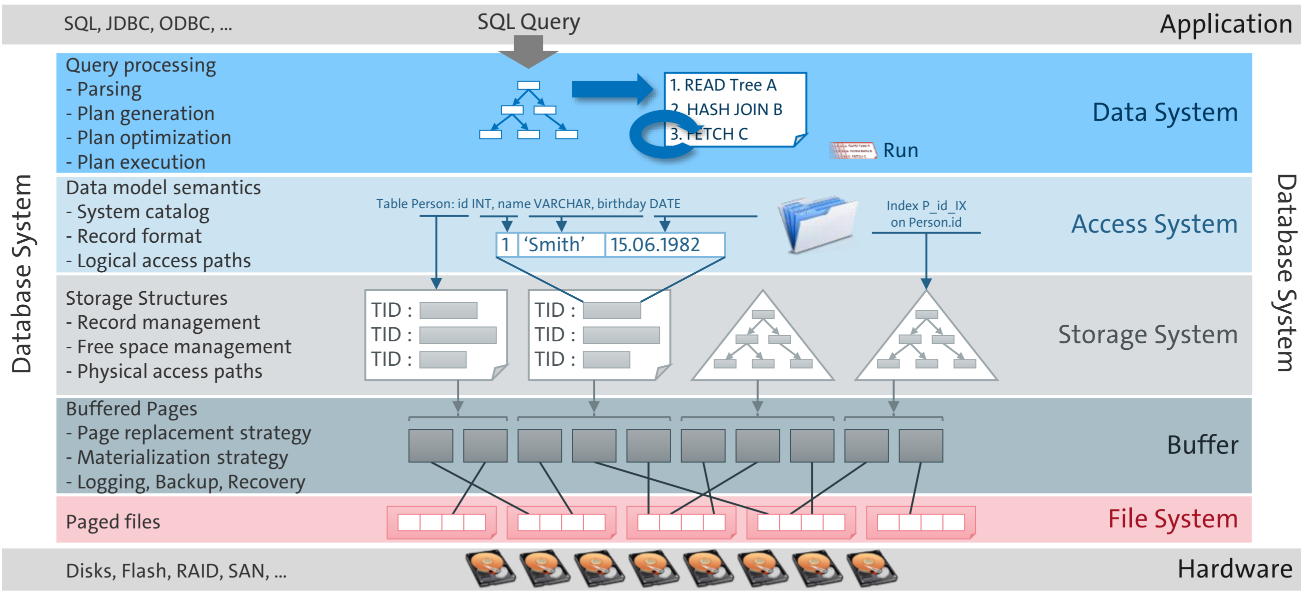
...and some show a formatted version of their internal RA representation (e.g. MonetDB, PostgreSQL)

$\sigma_{C\_5 < 3}(X\_15)$

```
function user.main():void;
    X_1:void := querylog.define("explain select count(*) from (select 1 from testsequence, authors where
);
    X_4:int := sql.mvc();
    C_5:bat[:oid] := sql.tid(X_4:int, "sys"                 ;
    X_8:bat[:str] := sql.bind(X_4:int, "sys":str, "testsequence":str, "pdb_id":str, 0:int);
    X_15:bat[:int] := sql.bind(X_4:int, "sys":str, "testsequence":str, "sequence_count":str, 0:int);
    C_22:bat[:oid] := algebra.thetaselect(X_15:bat[:int], C_5:bat[:oid], 3:int, "<":str);
    C_24:bat[:oid] := sql.tid(X_4:int, "sys":str, "authors":str);
    X_26:bat[:str] := sql.bind(X_4:int, "sys":str, "authors":str, "IDCODE":str, 0:int);
    X_31:bat[:str] := sql.bind(X_4:int, "sys":str, "authors":str, "AUTHOR":str, 0:int);
    X_36:bat[:str] := algebra.projection(C_22:bat[:oid], X_8:bat[:str]);
    X_38:bat[:str] := algebra.projection(C_24:bat[:oid], X_26:bat[:str]);
    X_41:bat[:oid] := algebra.join(X_38:bat[:str], X_36:bat[:str], nil:BAT, nil:BAT, false:bit, nil:lng)
    X_49:bat[:str] := algebra.projectionpath(X_41:bat[:oid], C_24:bat[:oid], X_31:bat[:str]);
    (X_50:bat[:oid], C_51:bat[:oid]) := group.groupdone(X_49:bat[:str]);
    X_53:bat[:str] := algebra.projection(C_51:bat[:oid], X_49:bat[:str]);
    X_56:bat[:bte] := algebra.project(X_53:bat[:str], 1:bte);
    X_57:lng := aggr.count(X_56:bat[:bte]);
    X_59:int := sql.resultSet(".%2":str, "%2":str, "bigint":str, 64:int, 0:int, 7:int, X_57:lng);
end user.main;
```

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

The 5 Layer Model



Transactions

- Transactions change the state of a database
  - → Records, TIDs, indexes,… must be changed and brought into a consistent state
  - → Transactions have certain properties to ensure this consistent state at the start and end of a transaction

→ **ACID**

- Transactions include one or more statements that change the state of the database, e. g. INSERT, UPDATE, DELETE, CREATE, DROP
- To execute a transaction, it must be committed (COMMIT)
- Many DB systems have an autocommit mode, CREATE and DROP is usually committed automatically regardless of the mode
- Start of a transaction with multiple statements: BEGIN TRANSACTION
- Abbreviations: BOT (Begin Of Transaction), EOT (End Of Transaction), DML (Data Manipulation Language)
- More useful commands for working with transactions: ROLLBACK, SET TRANSACTION, SAVEPOINT

ACID Properties

- **Atomicity**
  → "All or nothing" property of any DBMS action
- **Consistency and semantic integrity**
  → A successful transaction guarantees that all integrity requirements are met
- **Isolated execution**
  → "Logical single user mode"
- **Durability**
  → Requires that modified data of successful transactions must survive any type of failure

Requires Transaction Management, i.e. Synchronization and Recovery

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- Atomicity: Transactions are not split. Either the whole transaction is performed, or the transaction is not performed at all. There is no partial (successful) execution.
- Consistency: e.g. primary keys must be unique, values must be within their assigned value range, custom definitions (e.g. a product cannot have been sold before it was produced)
- Isolation: Multiple transactions are isolated from each other, i.e. they do not use inconsistent intermediate results of each other
- Durability: Successful transactions must be made persistent (unless the database is located entirely in volatile memory, then the result of the transaction is only written to this volatile memory)

Course Outline

- Architectures of Database Systems
- Transaction Management
  - Synchronization
  - Logging
  - Recovery
- Modern Database Technology
- Data Warehouses and OLAP
- Data Mining
- Big Data Analytics

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Why We Need Synchronization

Transaction 1

```
SELECT SALARY INTO :salary
FROM EMPL
WHERE ENR = 2345

salary := salary + 2000;

UPDATE EMPL
SET SALARY = :salary
WHERE ENR = 2345
```

Transaction 2

```
SELECT SALARY INTO :salary
FROM EMPL
WHERE ENR = 2345

salary := salary + 1000;

UPDATE EMPL
SET salary = :salary
WHERE ENR = 2345
```

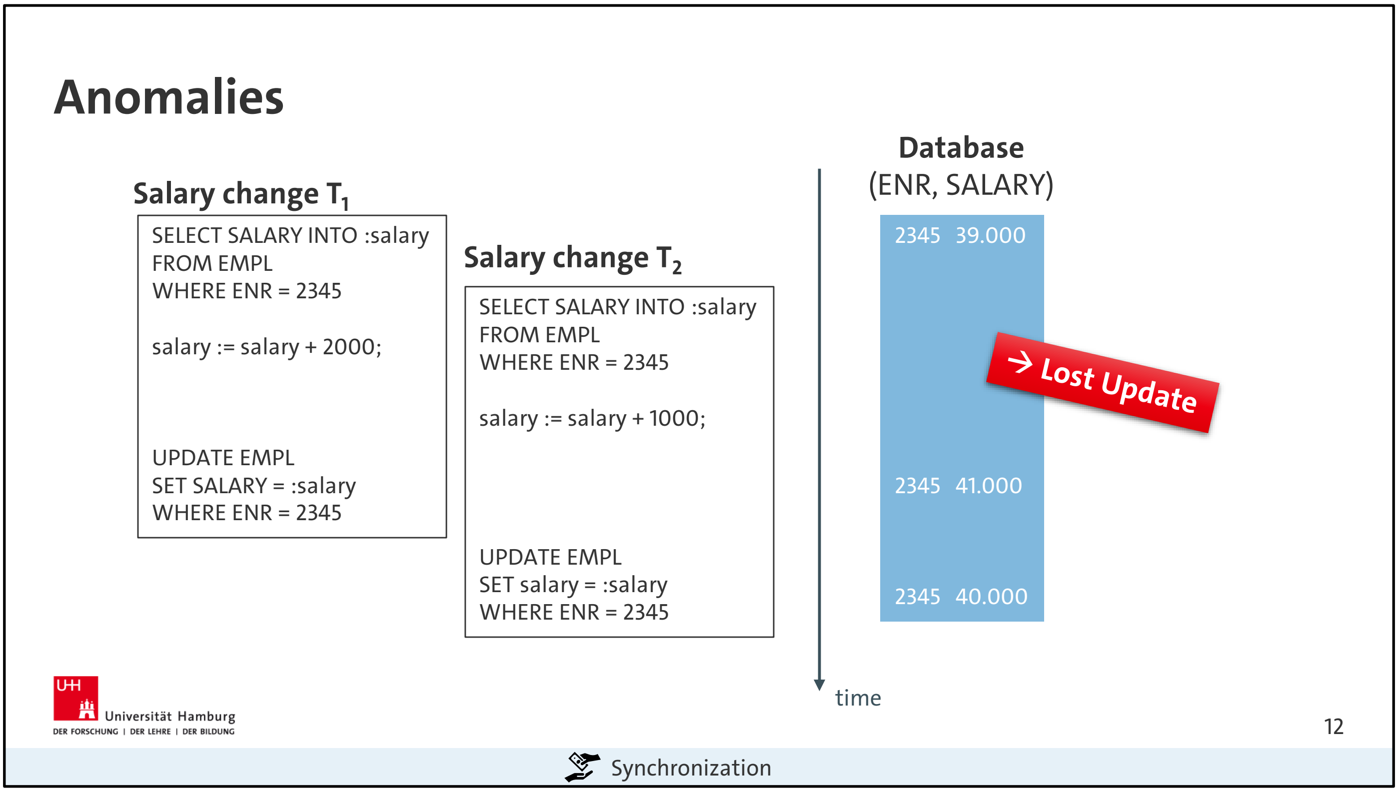
Which result so we expect after both transactions have finished?

Which results can we get if both transactions run concurrently?

→ If we let the transactions execute whenever they want, things can (and will) go wrong

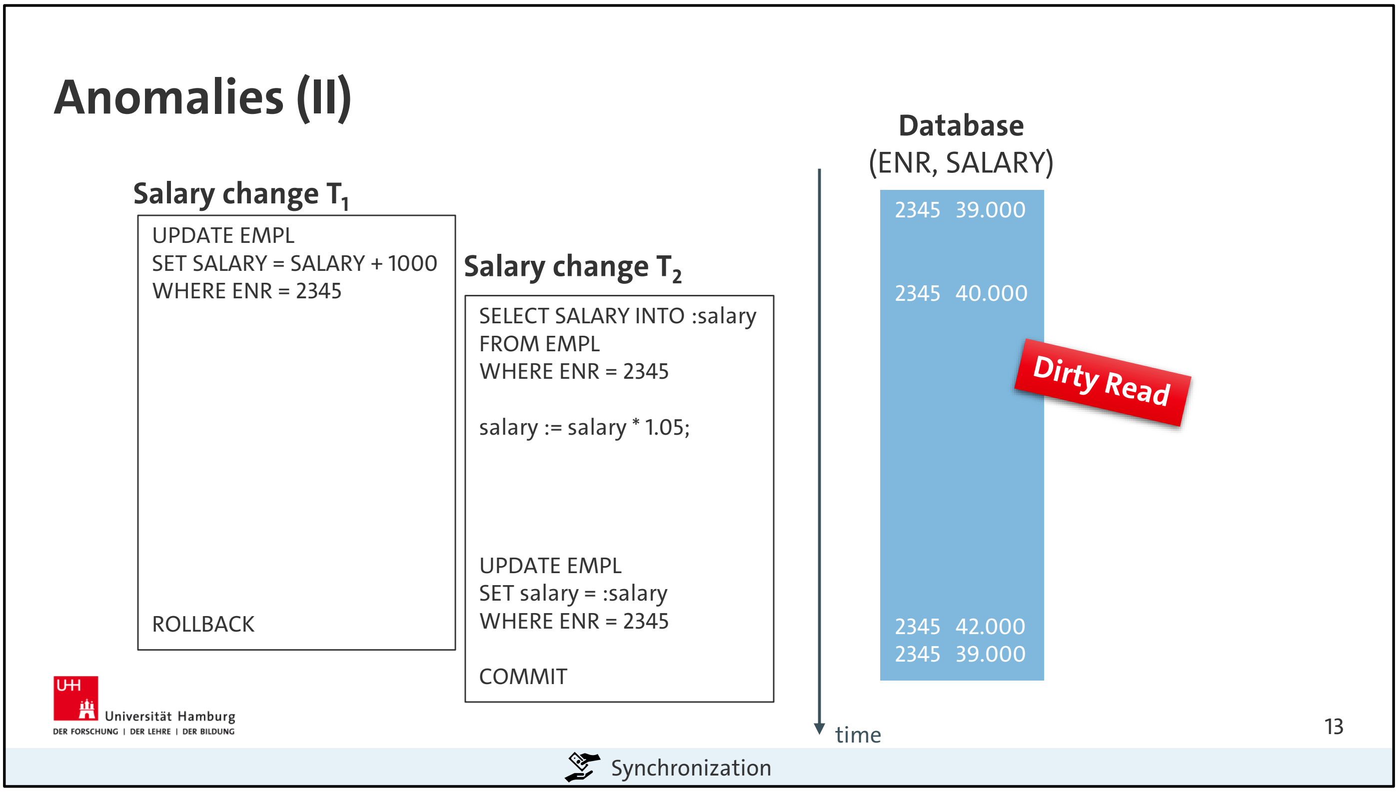
- A number of different anomalies can occur during concurrent execution of transactions
- We expect a salary increase of 3000, but this is not guaranteed if we let both transactions run concurrently without further measures

- Why do we not just run all transactions serially?
    - It’s slow!
    - It makes sense in many scenarios, e.g.:
        - Multiple Users (real and virtual)
        - Multiple available CPUs, distributed systems in general



- Both transactions read salary = 39.000
- The first transaction computes salary = 39.000 + 2.000 = 41.000
- The first transaction commits salary = 41.000
- The second transaction computes salary = 39.000 + 1.000 = 40.000
- Transaction 2 overwrites the result of transaction 1

- Possible solutions:
  - Locking, i.e. do not allow another transaction to write salary
  - Validate at the time of writing the update



- Transaction 2 reads a value that is not committed
- Transaction 1 had not finished while transaction 2 was already reading the changed value
  - → Database was not in a consistent state when T_2 started
- Solutions:
  - Read data only after is has been committed
- Dirty read is also sometimes referred to as *Inconsistent Read*

Anomalies (III)

Salary change T_1

```
UPDATE EMPL
SET SALARY = SALARY + 1000
WHERE ENR = 2345
UPDATE EMPL
SET SALARY = SALARY + 2000
WHERE ENR = 3456
COMMIT
```

Get salaries T_2

```
SELECT SALARY INTO :g1
FROM EMPL
WHERE ENR = 2345

SELECT SALARY INTO :g2
FROM EMPL
WHERE ENR = 3456

sum := g1 + g2
```

time

Database
(ENR, SALARY)

2345	39.000
3456	45.000
2345	40.000
3456	47.000

Non-Repeatable Read

What is the result for *sum* and which result did we expect?

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- Values change while T_2 is processed
- Database was in a consistent state during the start of T_2 and during the second read access of *SALARY*
- Solutions:
    - Lock read access
    - Multiversion concurrency control (mvcc) → DB holds multiple versions with time stamps or transaction numbers → later in this lecture

Anomalies (IV)

Get salaries T_1

```
SELECT SUM(Salary)
INTO :Sum1
FROM Empl
WHERE DepNr = 17
```

```
SELECT SUM(Salary)
INTO :Sum2
FROM Empl
WHERE DepNr = 17
```

Create new record T_2

```
INSERT INTO Empl(ENR, DepNr, Salary)
Values( 4567, 17, 55.000)
COMMIT
```

Database
(ENR, DepNr, Salary)

…		
2345	17	39.000
3456	17	45.000
…		

Sum 84.000

…		
2345	17	39.000
3456	17	45.000
…		
4567	17	55.000

Phantom Problem

139.000

time

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

15

Synchronization

- T1 computes the sum of *Salary* twice → the results are different
- Resembles a non-repeatable read, but spans multiple records or even the whole relation

Schedules (I)

Salary change T_1

```
SELECT SALARY INTO :salary
FROM EMPL
WHERE ENR = 2345

salary := salary + 2000;

UPDATE EMPL
SET SALARY = :salary
WHERE ENR = 2345
```

Salary change T_2

```
SELECT SALARY INTO :salary
FROM EMPL
WHERE ENR = 2345

salary := salary + 1000;

UPDATE EMPL
SET salary = :salary
WHERE ENR = 2345
```

Step	T_1	T_2
1	r(S)	
2		r(S)
3	w(S)	
4		w(S)

→ Lost Update

- A schedule shows the order in which the operations of different transactions are executed
- Schedules can be used to identify conflicts
- Possible operations are:
    - r – read
    - w – write
    - a – abort
    - c – commit

Schedules (II)

Salary change T_1

```
UPDATE EMPL
SET SALARY = SALARY + 1000
WHERE ENR = 2345

ROLLBACK
```

Salary change T_2

```
SELECT SALARY INTO :salary
FROM EMPL
WHERE ENR = 2345

salary := salary * 1.05;

UPDATE EMPL
SET salary = :salary
WHERE ENR = 2345

COMMIT
```

Step	T_1	T_2
1	w(S)	
2		r(S)
3		w(S)
4	a	
5		c

→ **Dirty Read**

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Example Schedule

Step	T_1	T_2	T_3	Step	T_1	T_2	T_3
1	r(A)			8		w(C)	
2		r(B)		9		w(A)	
3		r(C)		10			r(A)
4		w(B)		11			r(C)
5	r(B)			12	w(B)		
6	w(B)			13			w(C)
7		r(A)		14			w(A)

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

- r – read, w – write, (X) – object *X*
- DR – Dirty Read, LU – Lost Update

- “<“ is an ordering relation which is defined by the schedule

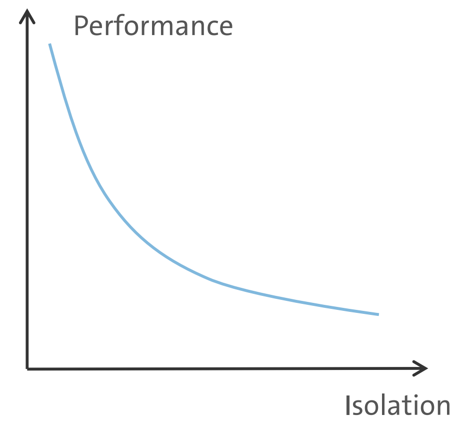
- Conflicts leading to anomalies, i.e. the sequence of execution is critical:
  - $w_2(B)<r_1(B)$ → DR
  - ~~$w_1(A)<r_2(A)$ → DR~~ (changed 08.04.24)
  - $w_2(C)<r_3(C)$ → DR
  - $w_2(A)<r_3(A)$ → DR
  - ~~$w_1(A)<w_2(A)$ → LU~~ (changed 08.04.24)
  - $w_2(A)<w_3(A)$ → LU
  - $w_2(B)<w_1(B)$ → LU (2x)
  - $w_2(C)<w_3(C)$ → LU

→ If multiple operations work with the same object and at least one of them is a write operation, a concurrent execution leads to conflicts

- The database is not in a consistent state during repeated read accesses. Thus, we do not have to search for non-repeatable reads.
- Phantom problem is not identifiable with the provided information

Isolation in SQL

- Isolation is expensive because it introduces locks
  →Transactions have to wait for each other
- Absolute Serializability (=avoiding anomalies) is not always necessary
  → SQL allows different isolation levels



- It’s always a trade-off
- Right level of isolation depends on the use-case

ANSI-SQL isolation levels

Higher isolation / protection against anomalies

Isolation Level	Lost Update possible?	Dirty Read possible?	Non-Repeatable Read possible?	Phantom Read possible?
Read Uncommitted	No	Yes	Yes	Yes
Read Committed	No	No	Yes	Yes
Repeatable Read	No	No	No	Yes
Serializable	No	No	No	No

More concurrency / higher performance

Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Set isolation level with SQL:
SET TRANSACTION ISOLATION LEVEL
{ READ UNCOMMITTED| READ COMMITTED |
REPEATABLE READ | SERIALIZABLE }

Special case: read uncommitted → Should avoid Lost Update, but does not always avoid them in reality

A more in-depth discussion and critique of the isolation levels can be found in the following paper:
Berenson, Hal, et al. "A critique of ANSI SQL isolation levels." *ACM SIGMOD Record* 24.2 (1995): 1-10. (https://dl.acm.org/doi/pdf/10.1145/568271.223785)
→ The list of authors might make it look heavily biased, which it might be. But the authors still have a point, and the paper is peer-reviewed