CS-300: Data-Intensive Systems

Transaction Management
(Chapters 17, 18)

*Prof. Anastasia Ailamaki, **Prof. Sanidhya Kashyap***

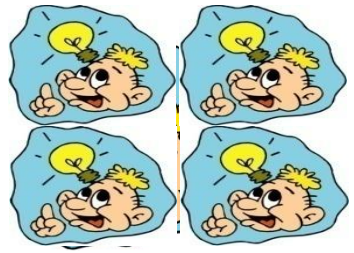
EPFL

Today's focus

- Overview of ACID
- Concurrency
- Logging and recovery

The big picture

Want to store data



Want to access data

Relational Algebra, SQL

SQL

Result

Relational Model

Conceptual Design

ER Models

ER to Relational

Logical Design

Query Optimization and Execution

Relational Operators

Access Methods

Buffer Management

Disk Space Management

Query processing

Transaction management

Concurrency control

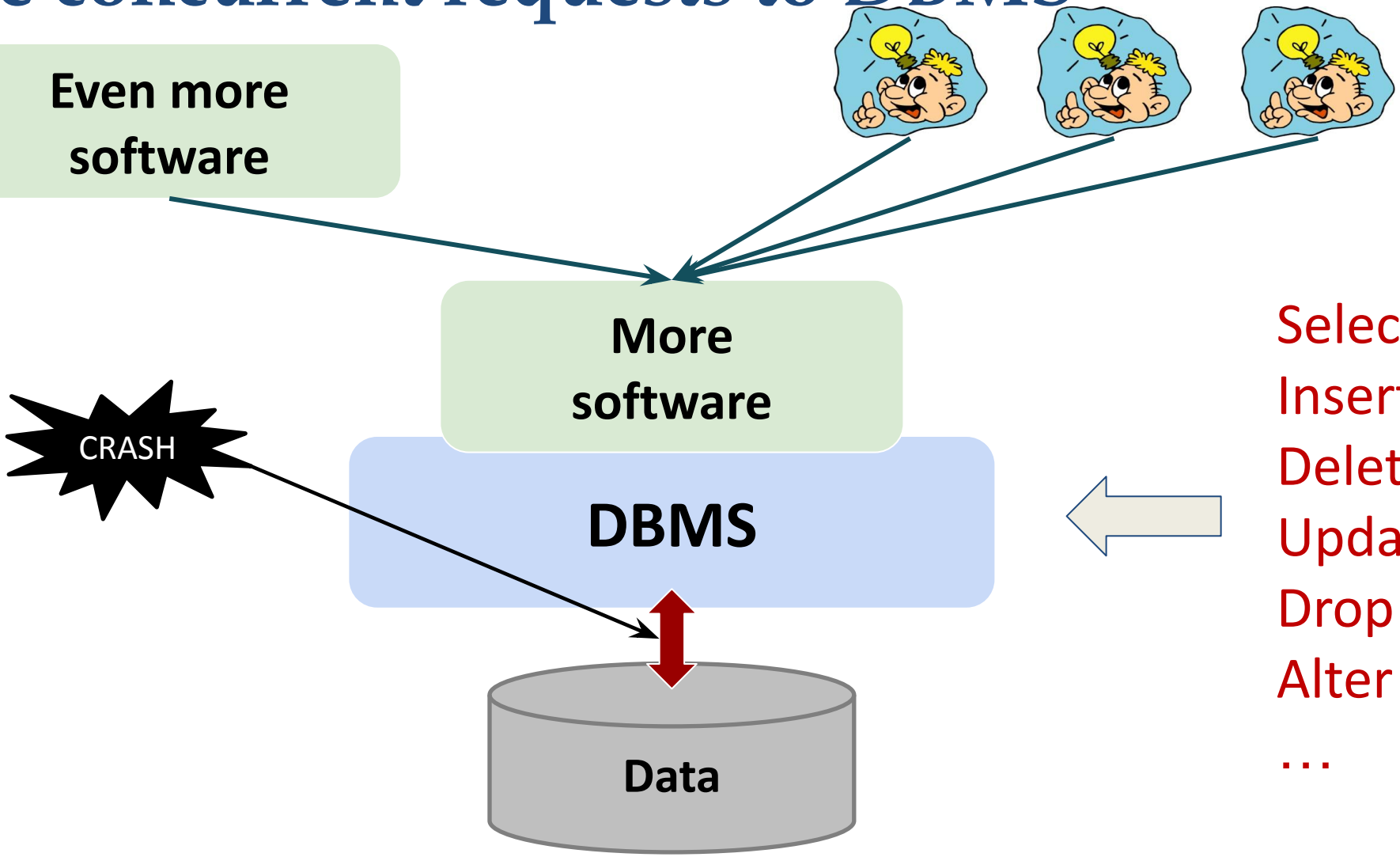
Recovery

Storage

Files

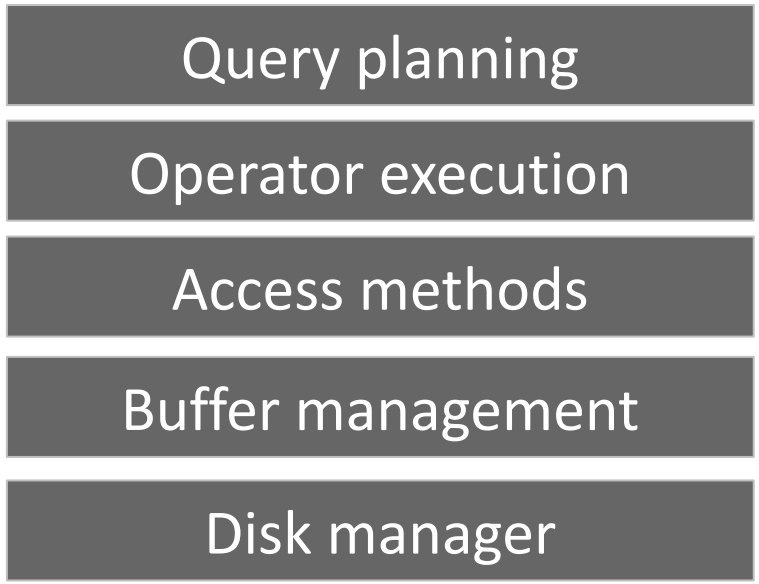
Q. Why is transaction management really important?

Multiple concurrent requests to DBMS



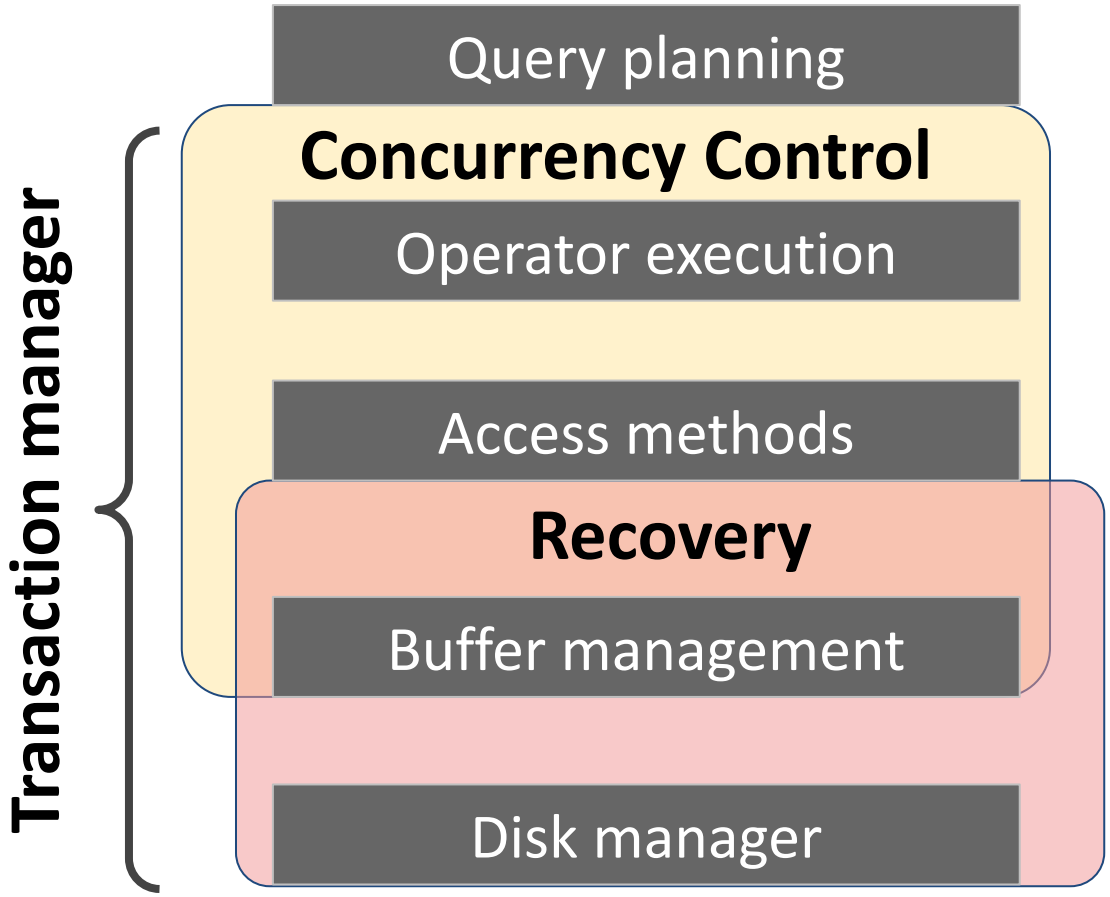
Transaction management is important across layers

→ Concurrency control and recovery components permeate throughout DBMS design



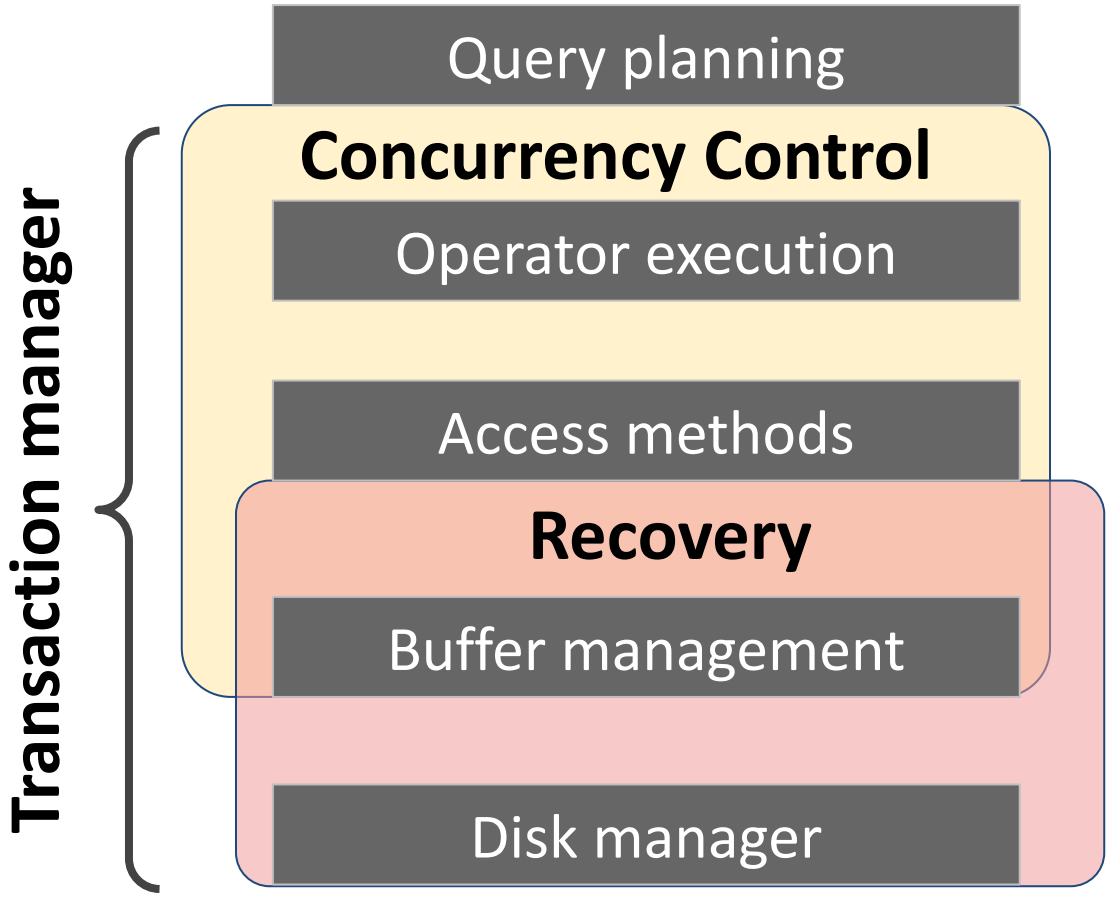
Transaction management is important across layers

→ Concurrency control and recovery components permeate throughout DBMS design



Transaction management ensures …

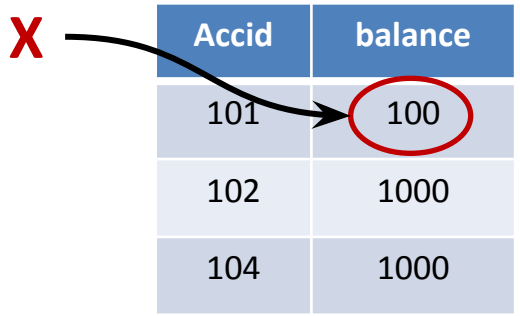
1. Concurrent database access (*concurrency control*)
2. Resilience to system failures (*recovery*)



1. Why concurrency becomes an issue?

Two concurrently executing queries

```
update account                  update account
set balance=balance-20          set balance=balance+10
where accid=101                 where accid=101
```



Read; modify; write

```
R1(X)                           R2(X)
X=X-20                          X=X+10
W1(X)                           W2(X)
```

```
t0: R1(X)
t1: X=X-20
t2: W1(X)
t3:          R2(X)
t4:          X=X+10
```

```
t0:          R2(X)
t1:          X=X+10
t2:          W2(X)
t3: R1(X)
t4: X=X-20
```

```
t0: R1(X)
t1:          R2(X)
t2: X=X-20
t3:          X=X+10
t4: W1(X)
```

Arbitrary interleaving can lead to inconsistencies

X=90 **X=90** **X=110**

1. Goal of concurrency control …

Execute a sequence of SQL statements so they appear to be running in isolation

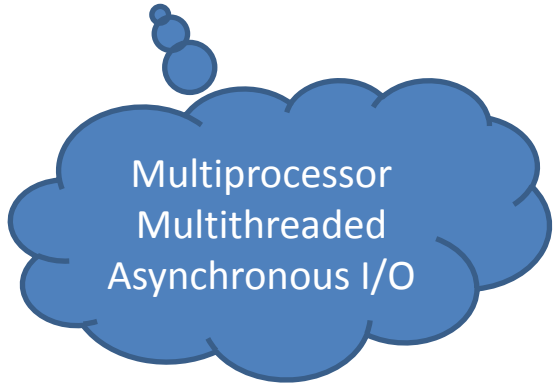
- **Strawman approach**: Execute each query one-by-one (i.e, serial order) as they arrive at the DBMS
  - One and only query runs simultaneously in the DBMS
- Before a query starts, copy the entire database to a new file and make all changes to that file
  - If the operation completes successfully, overwrite the original file with the new one
  - If the operation fails, just remove the dirty copy

Q. What is the issue with this approach?

1. Goal of concurrency control …

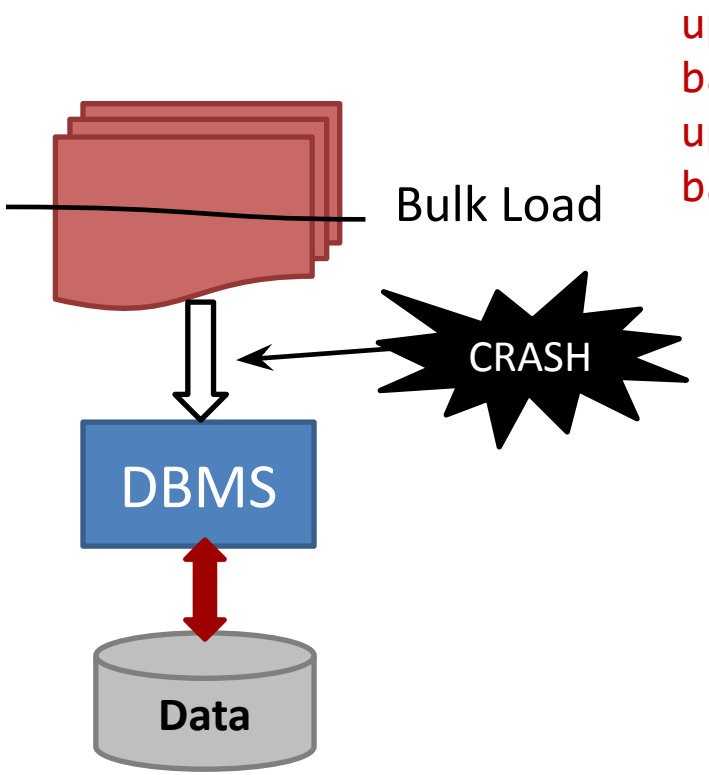
(Potentially) better approach**:** Allow concurrent execution of independent set of queries

- Why do we want?
  - Better utilization / throughput
  - Increased response time

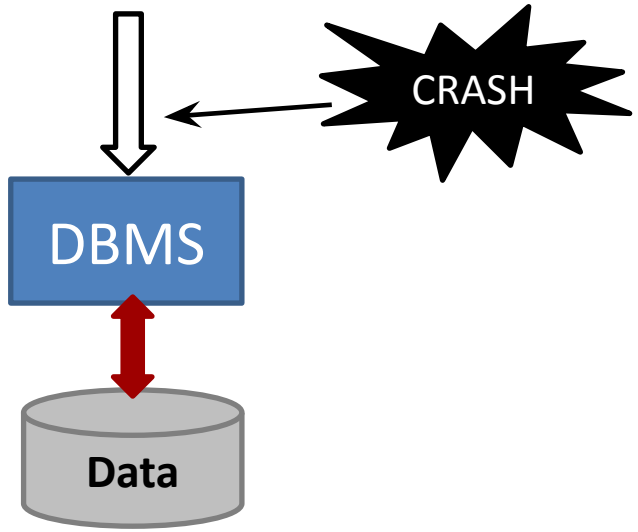


- But we would also like:
  - Correctness
  - Fairness

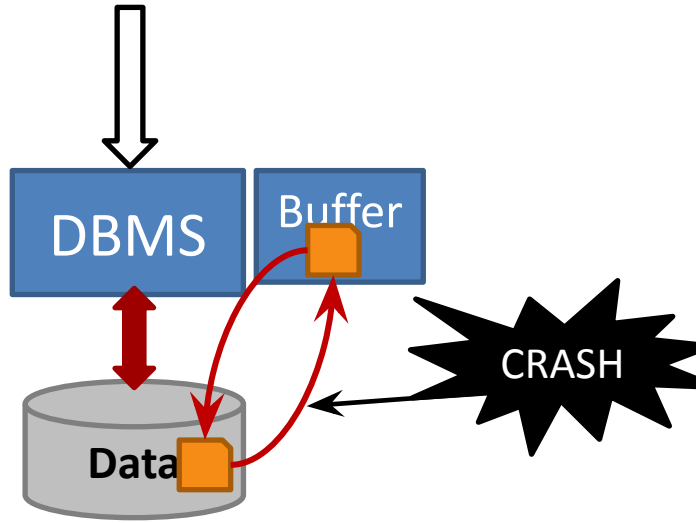
2. Resilience to system failures



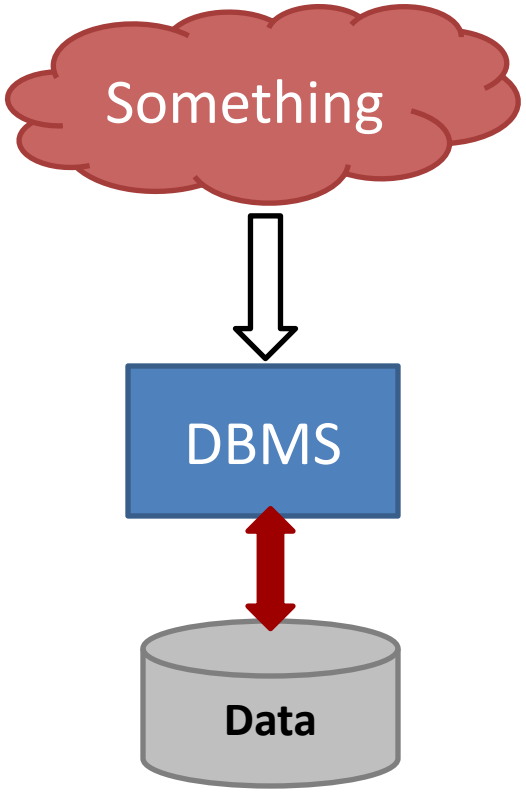

```
update account set
balance=balance-50 where accid=101
update account set
balance=balance+50 where accid=102
```



Lots of updates buffered in memory



2. Goal of handling system failures



Guarantee all-or-nothing execution, regardless of failures

Solution to both problems

- Concurrent database access
- Resilience to system failures

Transactions

- A sequence of one or more SQL operations treated as a unit
  - Transactions appear to run in isolation
  - If the system fails, the DBMS will either reflect the changes of the transaction or not at all

Using transactions efficiently and correctly

- (Potentially) better approach: Concurrently execute independent transactions

- Arbitrary interleaving of operations can lead to:
  - Temporary inconsistencies (ok, unavoidable)
  - Permanent inconsistency (bad!)

- The DBMS is only concerned about what data is read/written from/to the database
  - Changes to the “outside world” are beyond the scope of the DBMS

- We need a formal correctness criteria to determine whether an interleaving is valid!

Definitions

- A program may carry out many operations on the data retrieved from the database
- The DBMS is only concerned about what data is read/written from/to the database
  - Changes to the “outside world” are beyond the scope of the DBMS

Formal definition

- **Database:** A fixed set of named objects (e.g., **A**, **B**, **C**, …)
- **Transaction:** A sequence of read and write operations (**R(A)**, **W(B)**, …)

Transactions in SQL

- A new transaction starts with the **`BEGIN`** command
- The transaction stops with either **`COMMIT`** or **`ABORT`**:
  - If **commit**, the DBMS saves all the transaction's changes **or** aborts it
  - If **abort**, all changes are undone → seems like transactions never executed at all

- Aborts can be either self-inflicted or caused by the DBMS

Correctness criteria: The ACID properties

- **Atomicity**: All actions in the *transaction* happen, or none happen ("*All or nothing*")
- **Consistency**: If each transaction is consistent, and the DB starts consistent, it ends up consistent ("*It looks correct to me*")
- **Isolation**: Execution of one *transaction* is isolated from that of other *transactions* ("*All by myself*")
- **Durability**: If a transaction commits, its effects persist ("*I will survive*")

A

Atomicity of transactions

- Two possible outcomes of executing a transaction:
  - Commit after completing all its actions
  - Abort (or be aborted by the DBMS) after executing some actions

- DBMS guarantees that transactions are **atomic**
  - From the user's point of view: transaction always either executes all its actions or executes no actions at all

A

Atomicity of transactions

Scenario #1:

- We take CHF100 out of an account, but then the DBMS aborts the transaction before we transfer it

Scenario #2:

- We take CHF100 out of an account, but there is a power failure before we transfer it

Q. What should be the correct state of the account after both transactions abort?

A

Mechanisms for ensuring atomicity

Approach #1: Logging

- DBMS logs all actions so that it can **undo** the actions of aborted transactions
- Maintain undo records both in memory and on disk
- Think of this like the black box in airplanes…

Logging is used by almost every DBMS

- Audit trails
- Efficiency reasons

Widely used approach!

A

Mechanisms for ensuring atomicity

Approach #2: Shadow paging

- DBMS makes copies of pages, and transactions makes changes to those copies
  - Only when the transaction commits, then page is made visible to others
  - Pros: Leads to instant recovery
  - Cons: Introduces memory fragmentation
- Originally from IBM System R

- Not prevalent in use; few systems still implement it:
  - CouchDB
  - Tokyo Cabinet
  - LMDB

A

Understanding transaction abort (i.e., rollback)

- If a transaction is aborted, all of its actions must be undone

- To undo actions of an aborted transaction, the DBMS *maintains a log* that records every write operation

- **Log** is used to recover DBMS from system crashes:
  - All active transactions at the time of crash are aborted when the system comes back up

The log

- Log consists of “records” that are written sequentially

  → Every change (insert, delete, update) becomes a log record appended to a file written in order

- Typically chained together by *transaction id*

  → Each record stores the ID of the previous record for the same txn for crash recovery by undoing operations backward

- Log is often *archived* on stable storage

- Records table, row information, also contain *old and new data*

  → Stores page-ID, offset, length, **before-image** for UNDO log and **after-image** for REDO log

The log

- **Write-ahead-logging (WAL)** protocol follows two rules
  1. **Ordering:** Log record must go to disk **before** updating the page
     - Implemented via a handshake between the log manager and the buffer manager

  2. **Commit boundary:** All log records for a transaction (including its commit record) must be written to disk before the transaction is considered "**committed**"

- DBMS handles all logging and concurrency control related protocols

C

Transaction consistency

- The DBMS accurately models the real world
- It also follows the integrity constraints (IC)
  - SQL has methods to define these constraints (e.g., key definitions, CHECK and ADD CONSTRAINT) and the DBMS enforces them
  - Applications must define these constraints
    - DBMS does not understand the semantics of data, relies on the application
    - Eg. it does not understand how to compute interest on a bank account
  - On failure of these constraints, the transaction rolls back (i.e., aborted)
  - DBMS ensures that all ICs are true before and after the transaction ends

Isolation of transactions

- Users submit transactions, and each transaction ensures as it were running by itself
  - Easier programming model to reason about

- The DBMS achieves concurrency by interleaving the actions (reads/writes of DB objects) of transactions

- We need a way to interleave transactions but still make it appear as if they ran **one-at-a-time**

Mechanisms for ensuring isolation

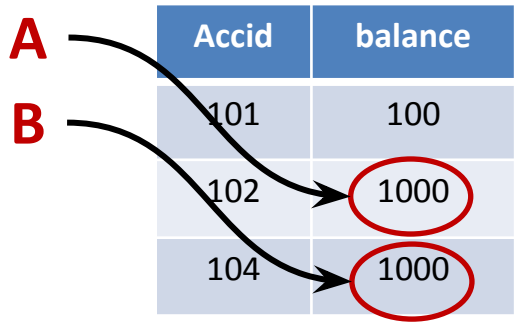
A **concurrency control** protocol is how the DBMS decides the proper interleaving of operations from multiple transactions

Two categories of protocols:

- **Pessimistic:** Don't let problems arise in the first place
- **Optimistic:** Assume conflicts are rare; deal with them after they happen

Example

- Assume at first **A** and **B** each have CHF 1000
- T_1 transfers CHF 100 from **A**'s account to **B**'s
- T_2 credits both accounts with 6% interest



T_1

```
BEGIN
A = A - 100
B = B + 100
COMMIT
```

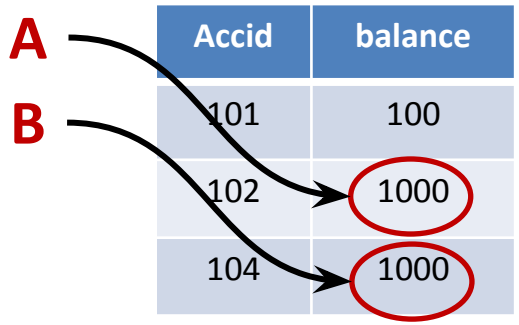
T_2

```
BEGIN
A = A * 1.06
B = B * 1.06
COMMIT
```

Example

- Assume at first **A** and **B** each have CHF 1000

What are the possible outcomes of running T_1 and T_2?



T_1

```
BEGIN
A = A - 100
B = B + 100
COMMIT
```

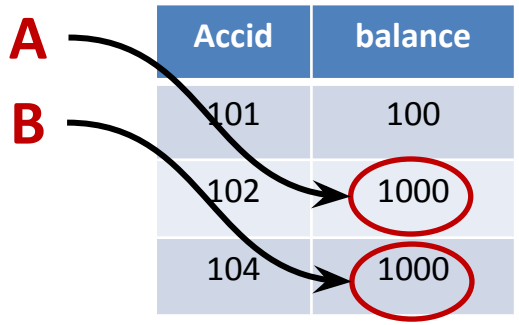
T_2

```
BEGIN
A = A * 1.06
B = B * 1.06
COMMIT
```

Example

- Assume at first **A** and **B** each have CHF 1000

What are the possible outcomes of running T_1 and T_2?



Many! But **A+B** should be:

- CHF 2000 * 1.06 = CHF 2120

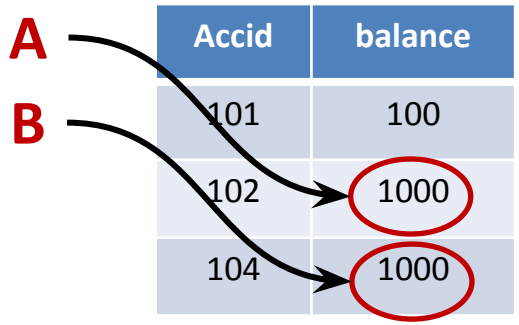
- There is no guarantee that T_1 will execute before T_2 or vice-versa, if both are submitted together

- But the net effect must be equivalent to these two transactions running **serially** in some order

Example

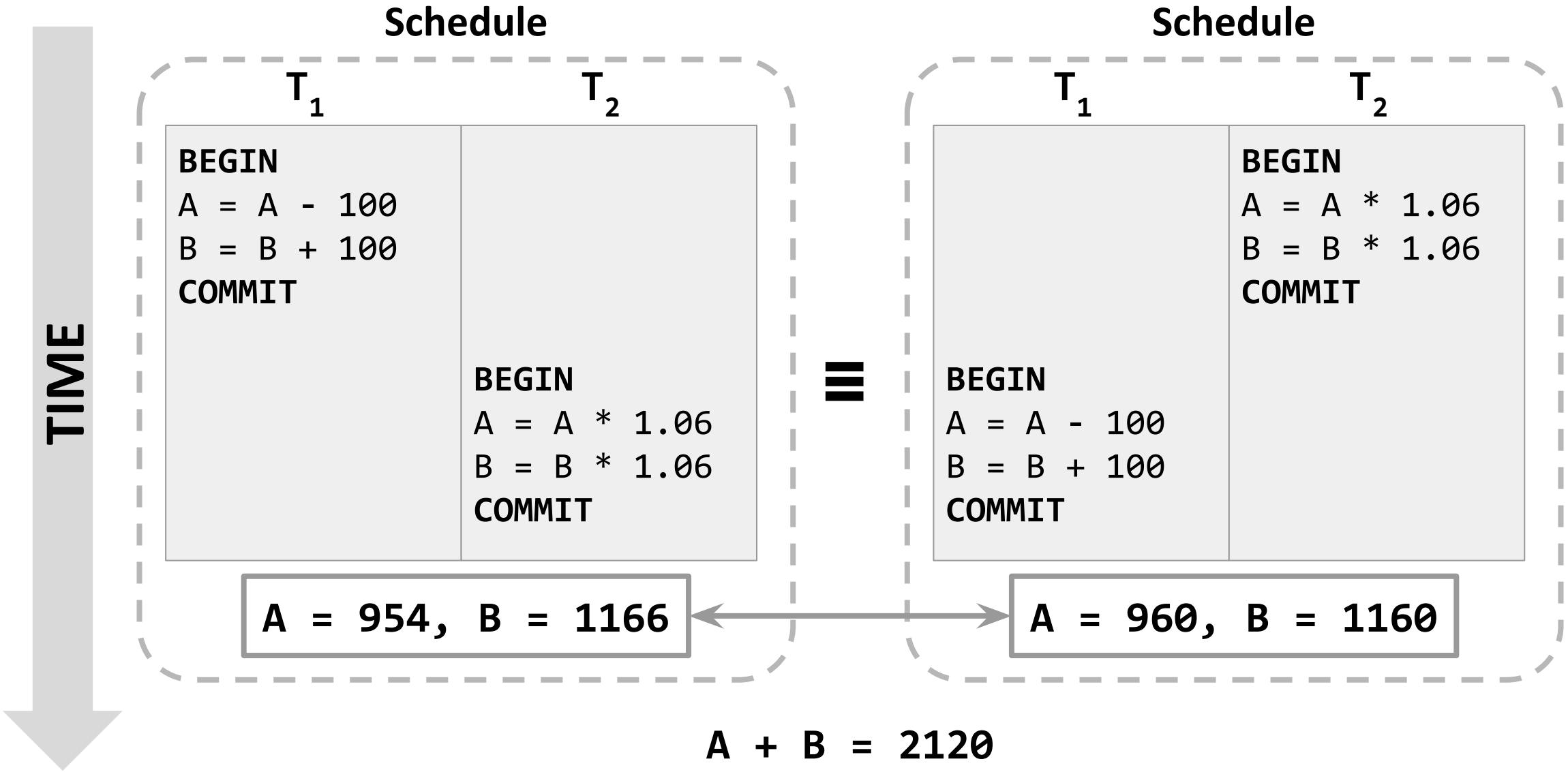
Legal outcomes:

- **A** = 954, **B** = 1166 → **A+B = 2120**
- **A** = 960, **B** = 1160 → **A+B = 2120**



The outcome depends on whether T_1 executes before T_2 or vice versa

Serial execution example

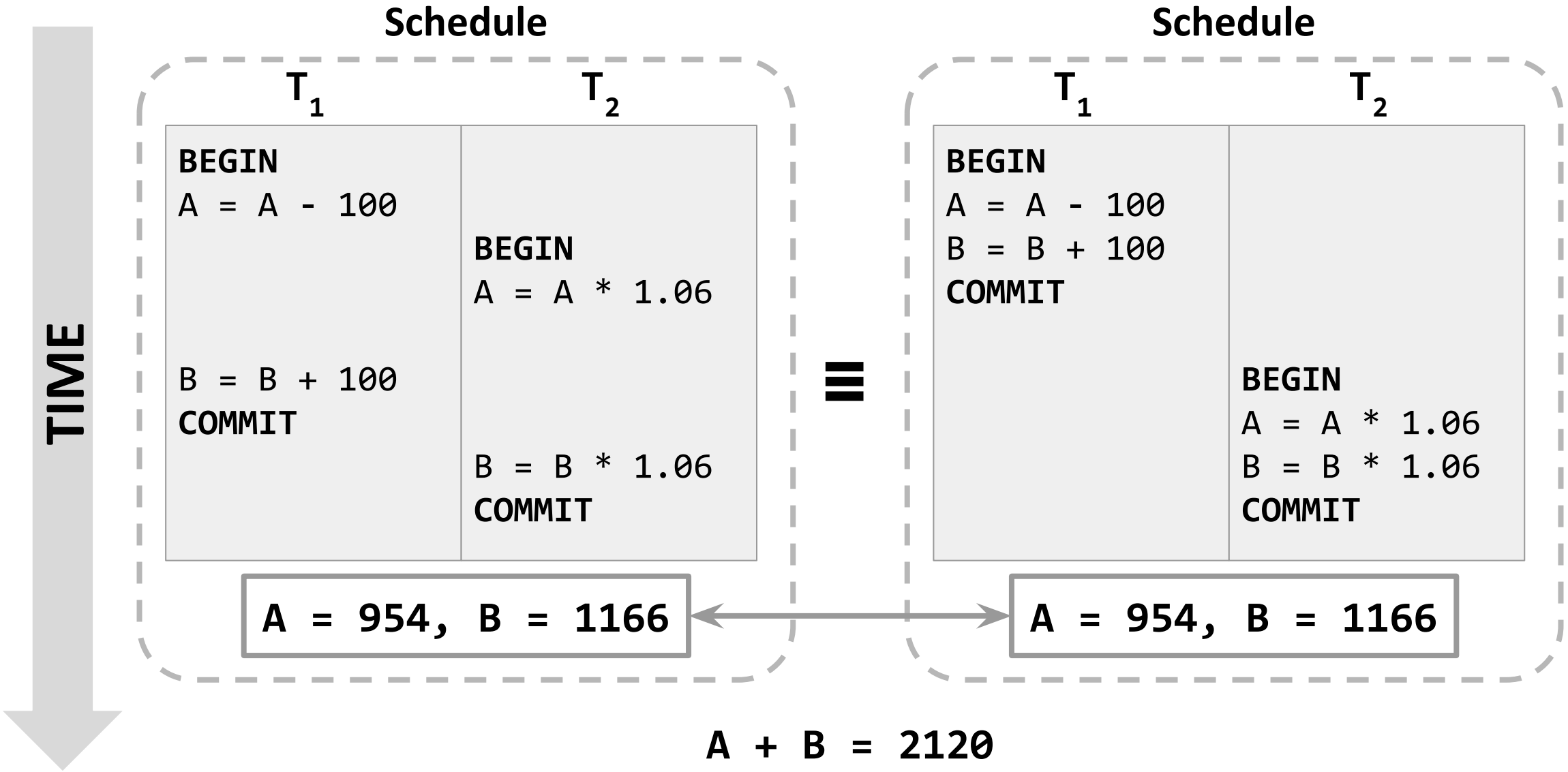


Interleaving transactions

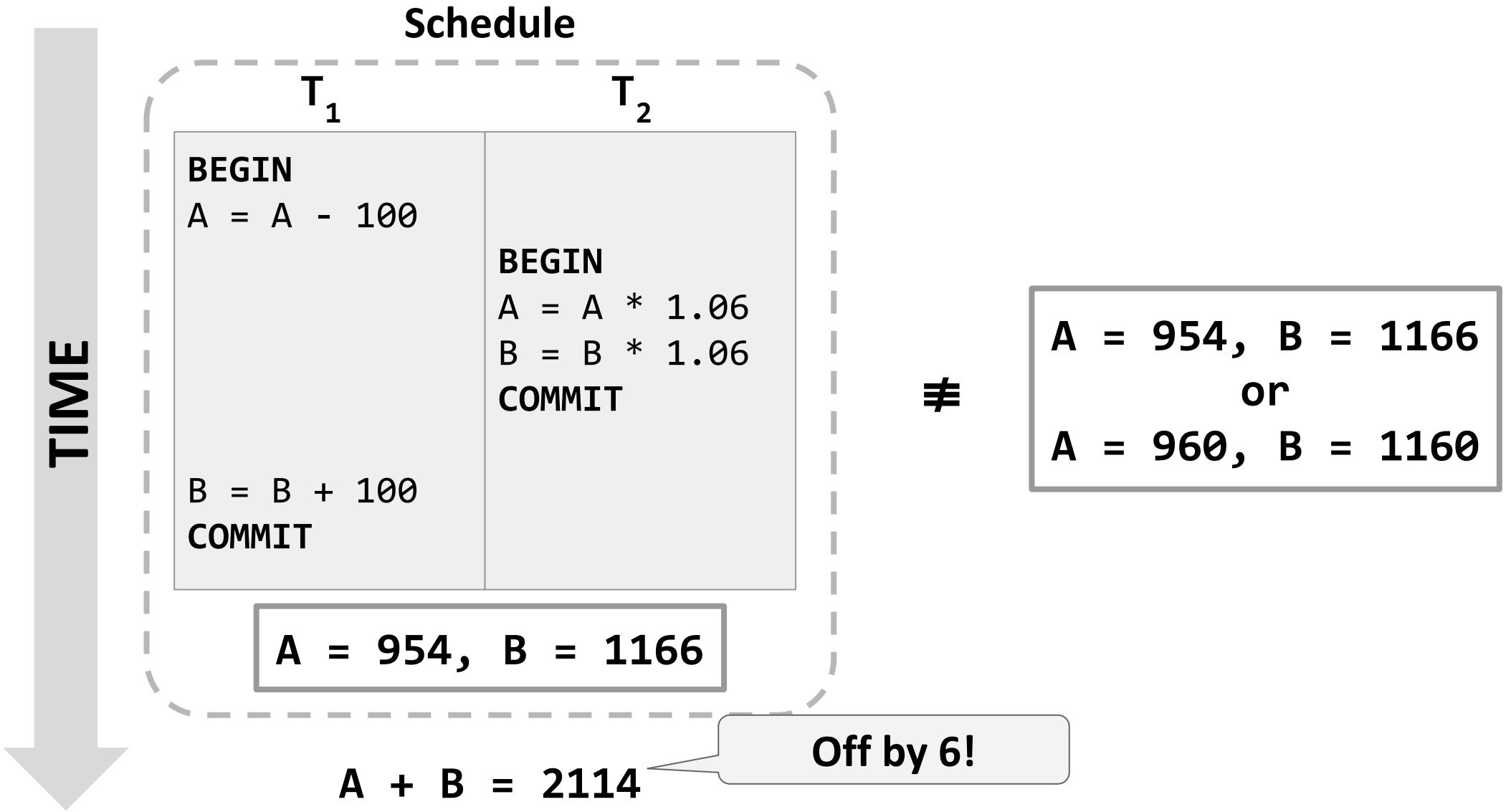
- We interleave transactions to maximize concurrency
  - Slow disk/network IOs
  - Multi-core CPUs

- When a transaction stalls because of a resource (e.g., page fault), another transaction can continue executing and make forward progress

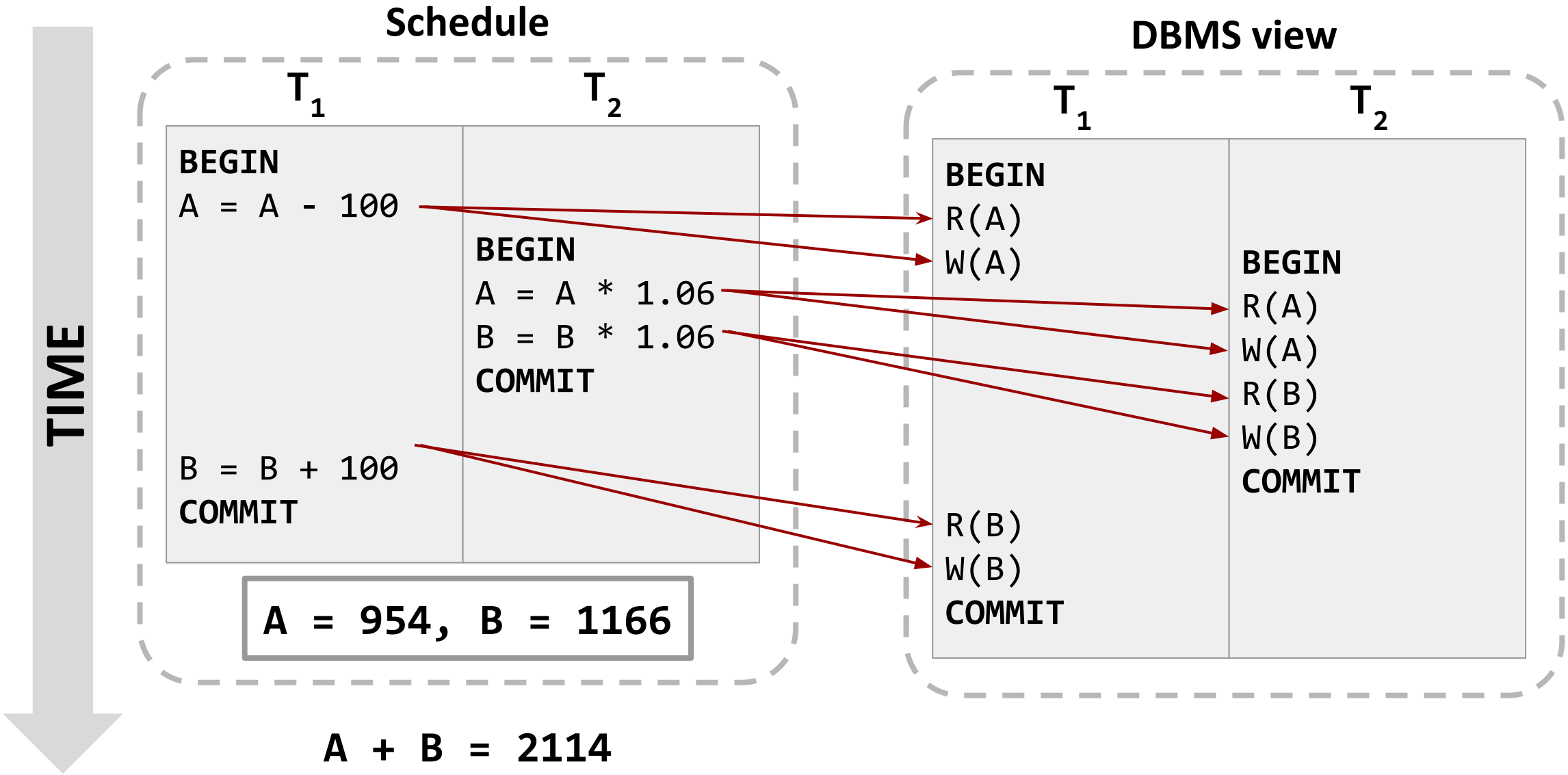
Interleaving example (good)



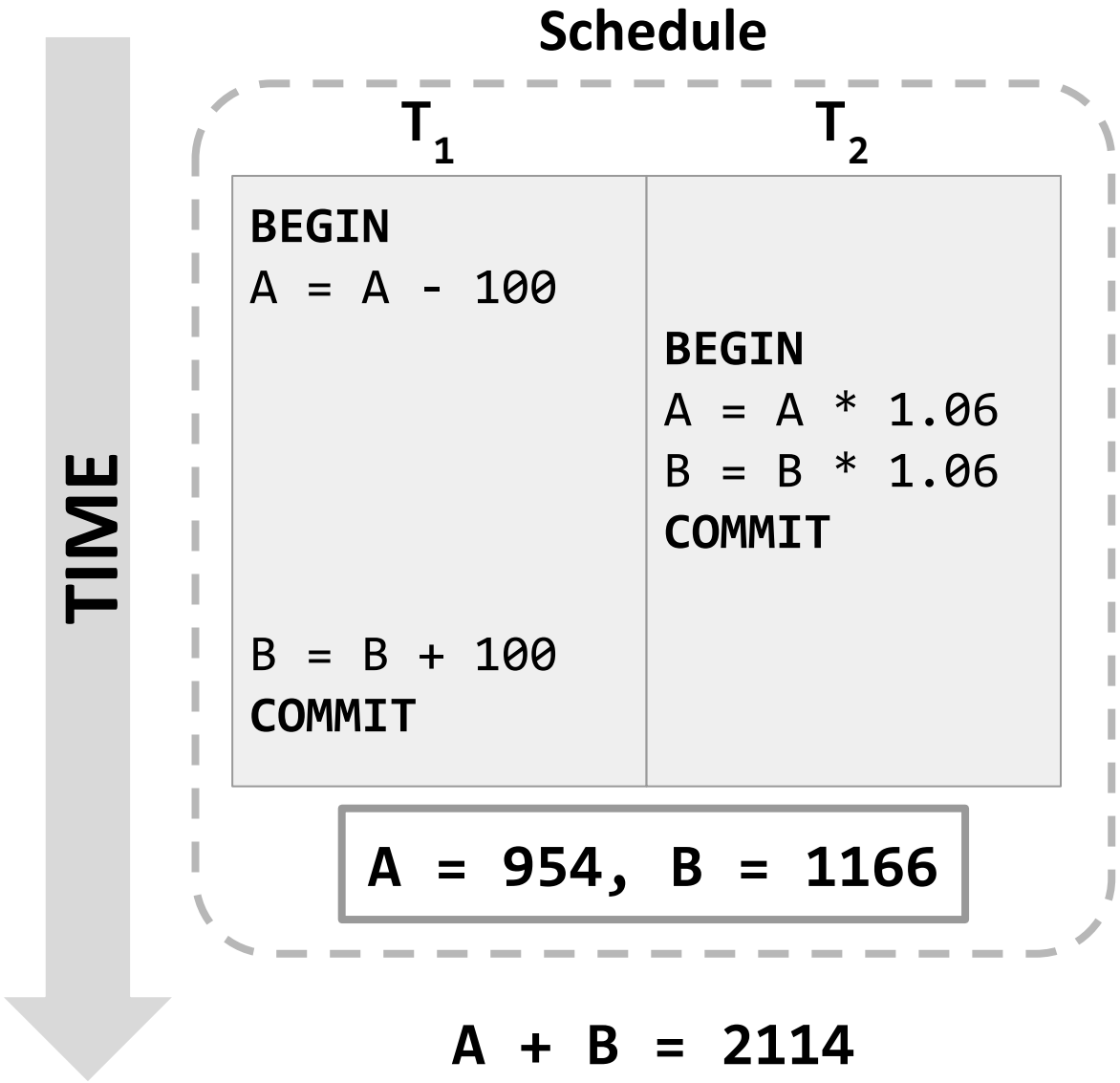
Interleaving example (bad)



Interleaving example (bad)



Interleaving example (bad)



How do we judge whether a schedule is correct?

If the schedule is **equivalent** to some **serial execution**

Formal properties of schedules

Serial schedule

- A schedule that does not interleave the actions of different transactions

Equivalent schedule

- For any database state, the effect of executing the first schedule is identical to the effect of executing the second schedule

Formal properties of schedules

Serializable schedule

- A schedule that is equivalent to some serial execution of the transactions
- If each transaction preserves consistency, every serializable schedule preserves consistency

Serializability is a less intuitive notion of correctness compared to transaction initiation time or commit order, but it provides the DBMS with more flexibility in scheduling operations

- More flexibility means more parallelism

Conflicting operations

Need a formal notion of equivalence that can be implemented efficiently based on the notion of “conflicting” operations:

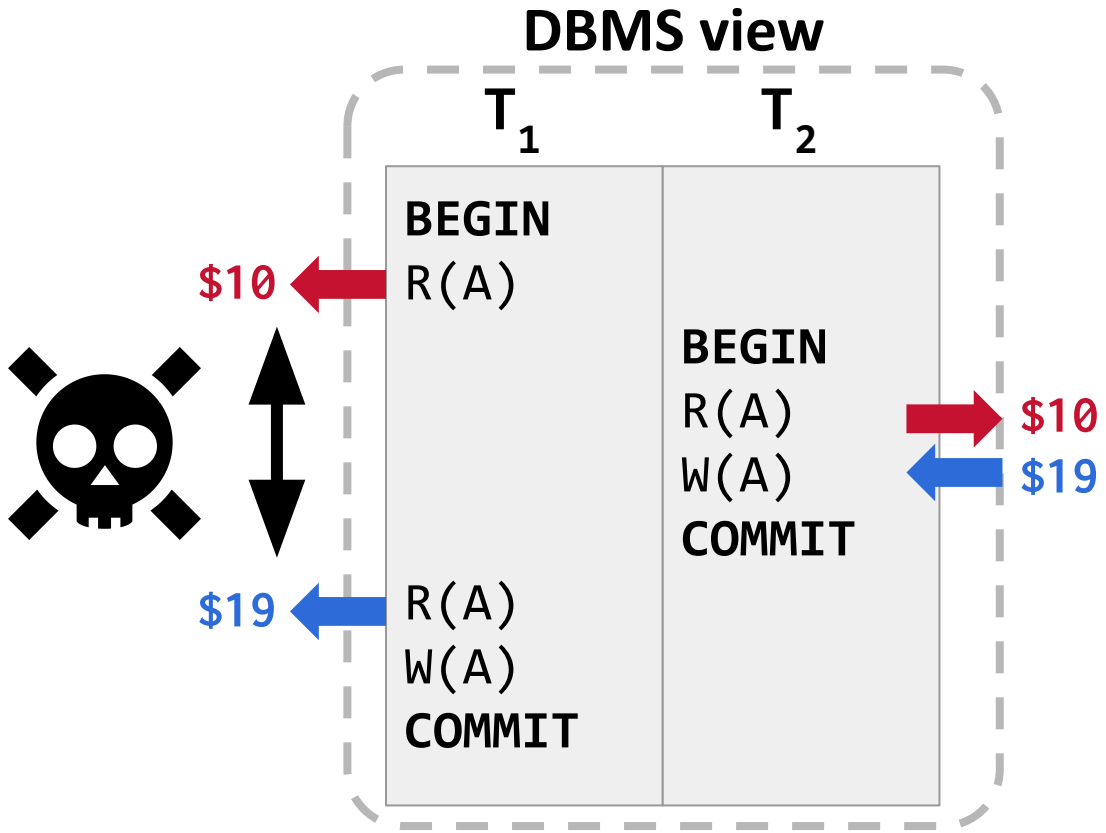
- Two operations conflict if:
  - They are by different transactions
  - They are on the same object and one of them is a write operation

Interleaved execution anomalies:

- Unrepeatable read (read-write)
- Dirty read (write-read)
- Lost update (write-write)
- Phantom reads (scan-write)
- Write skew (read-write)

Read-write conflicts

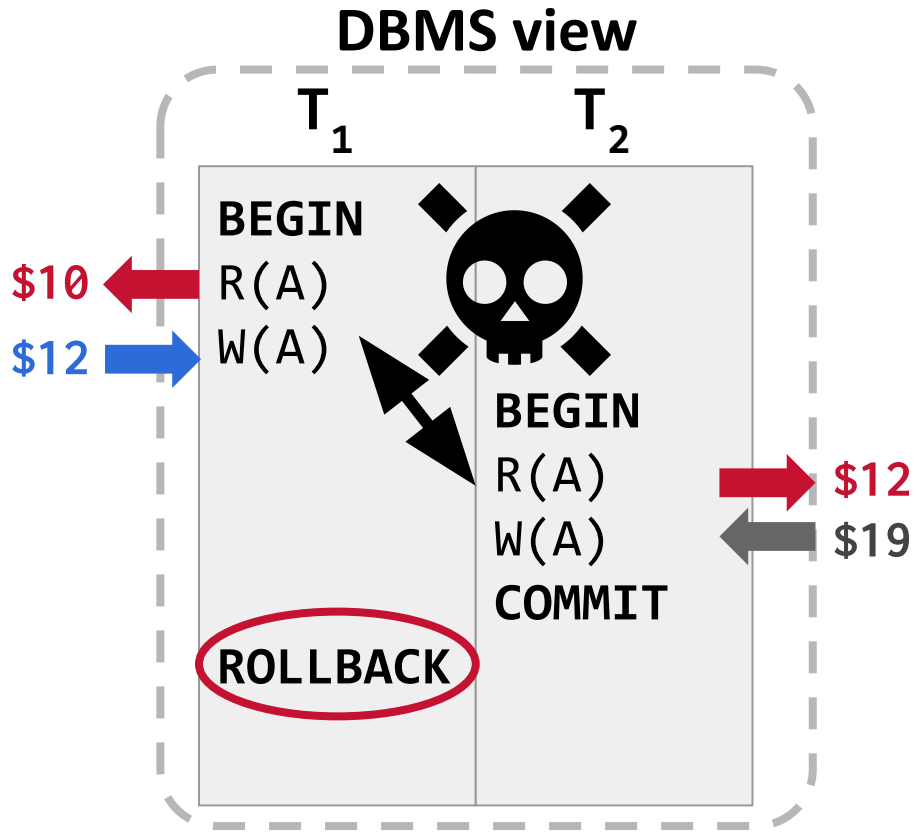
Unrepeatable read: Transaction gets different values when reading the same object multiple times



Violates isolation, not alone in the system

Write-read conflicts

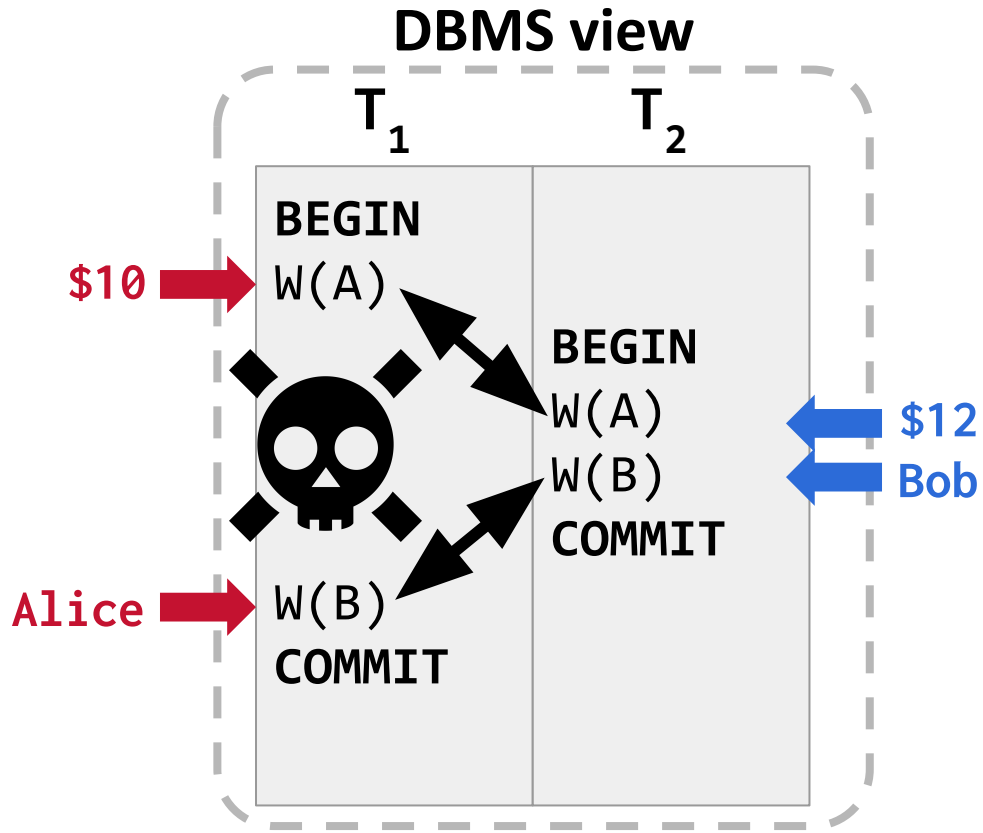
Dirty read: One transaction reads data written by another transaction that has not committed yet



Can read garbage value & not isolated

Write-write conflicts

lost update: One transaction overwrites uncommitted data from other uncommitted transaction



Addressing anomalies with lock-based CC

- Using locks is one of the simple ways to avoid such anomalies

Strict two-phase locking (S2PL) protocol

- Each transaction must obtain an **`S (shared)`** lock on the object before reading, and **`X (exclusive)`** lock on the object before writing
- System can obtain these locks automatically
- Must avoid deadlock situations by acquiring locks in a canonical order

Addressing anomalies with lock-based CC

- Lock rules:
  - If a transaction holds an **X** lock on an object, no other transaction can acquire a lock (**S** or **X**) on that object
  - If a transaction holds an **S** lock, no other transaction can get an **X** lock on that object
- Two phases: acquiring locks and releasing them
  - No lock is ever acquired after one has been released
  - All locks held by a transaction are released when the transaction completes

- S2PL avoids cascading aborts:
  - A situation in which the abort of one transaction forces the abort of another transaction to prevent the second transaction from reading invalid (uncommitted) data

D

Durability: Recovering from a crash

- All the changes of committed transaction should be persistent
  - No torn updates
  - No changes from failed transactions

- The DBMS can use either logging or shadow paging to ensure that all changes are durable

D

Durability: Recovering from a crash

- Three phases:
  - **Analysis:** Scan the log (forward from the most recent checkpoint) to identify all transactions that were active at the time of the crash
  - **Redo:** Redo updates as needed to ensure that all logged updates are in fact carried out and written to the disk
  - **Undo:** Undo writes of all transactions that were active at the time of the crash, working backwards in the log
- At the end: all committed updates and only those updates are reflected in the database
- Some care must be taken to handle the case of crash occurring during the recovery process

Correctness criteria: The ACID properties

Redo/Undo mechanism

- **Atomicity**: All actions in the *transaction* happen, or none happen ("All or nothing")

Integrity constraint

- **Consistency**: If each transaction is consistent, and the DB starts consistent, it ends up consistent ("It looks correct to me…")

Concurrency control

- **Isolation**: Execution of one *transaction* is isolated from that of other *transactions* ("All by myself …")

Redo/Undo mechanism

- **Durability**: If a transaction commits, its effects persist ("I will survive…")

Summary

- **Concurrency control** and **recovery** are among the most important functions by a DBMS
- Concurrency control is automatic
  - System automatically inserts lock/unlock requests and scheduled actions of different transactions
  - Property ensured: Resulting execution is equivalent to executing transactions one after the other in some order
- Write-ahead-log (WAL) and the recovery protocol are used to:
  - Undo the actions of aborted transactions, and
  - Restore the system to a consistent state after a crash