CS 525: Advanced Database Organization

12: Transaction Management

Boris Glavic

Slides: adapted from a course taught by Hector Garcia-Molina, Stanford InfoLab

Concurrency and Recovery

- DBMS should enable multiple clients to access the database concurrently
  - This can lead to problems with correctness of data because of interleaving of operations from different clients
  - ->System should ensure correctness (concurrency control)

Concurrency and Recovery

- DBMS should enable reestablish correctness of data in the presence of failures
  - ->System should restore a correct state after failure (recovery)

Integrity or correctness of data

- Would like data to be "accurate" or "correct" at all times

EMP

Name	Age
White	52
Green	3421
Gray	1

Integrity or consistency constraints

- Predicates data must satisfy
- Examples:
  - x is key of relation R
  - $x \rightarrow y$ holds in R
  - Domain(x) = {Red, Blue, Green}
  - α is valid index for attribute x of R
  - no employee should make more than twice the average salary

Definition:

- Consistent state: satisfies all constraints
- Consistent DB: DB in consistent state

Constraints (as we use here) may not capture "full correctness"

Example 1   Transaction constraints

- When salary is updated,
  new salary > old salary
- When account record is deleted,
  balance = 0

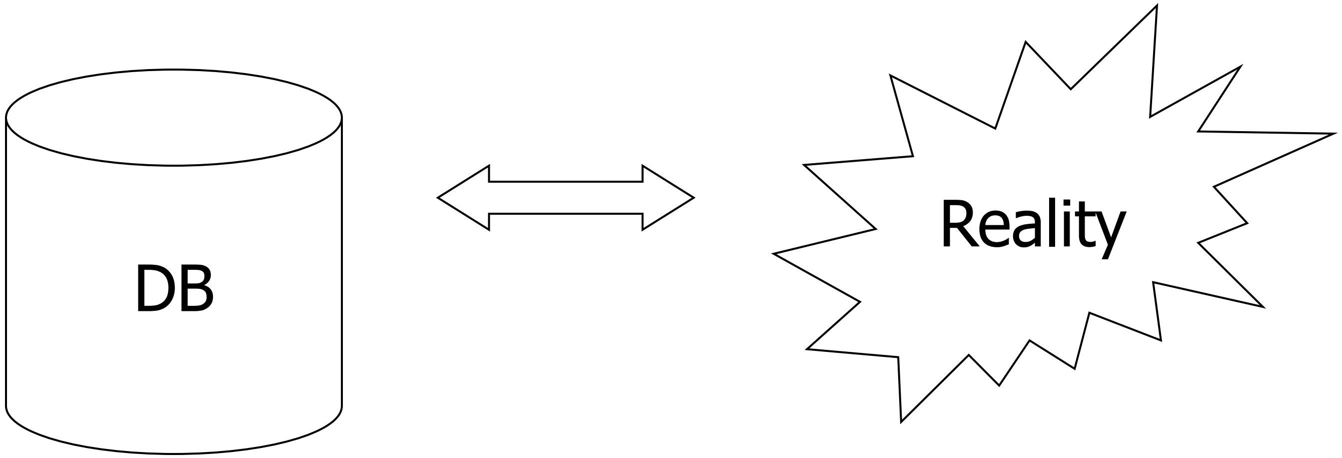
Note: could be “emulated” by simple constraints, e.g.,

account

Acct #	….	balance	deleted?

Constraints (as we use here) may not capture "full correctness"

Example 2 Database should reflect real world



☞ in any case, continue with constraints...

Observation: DB cannot be consistent always!

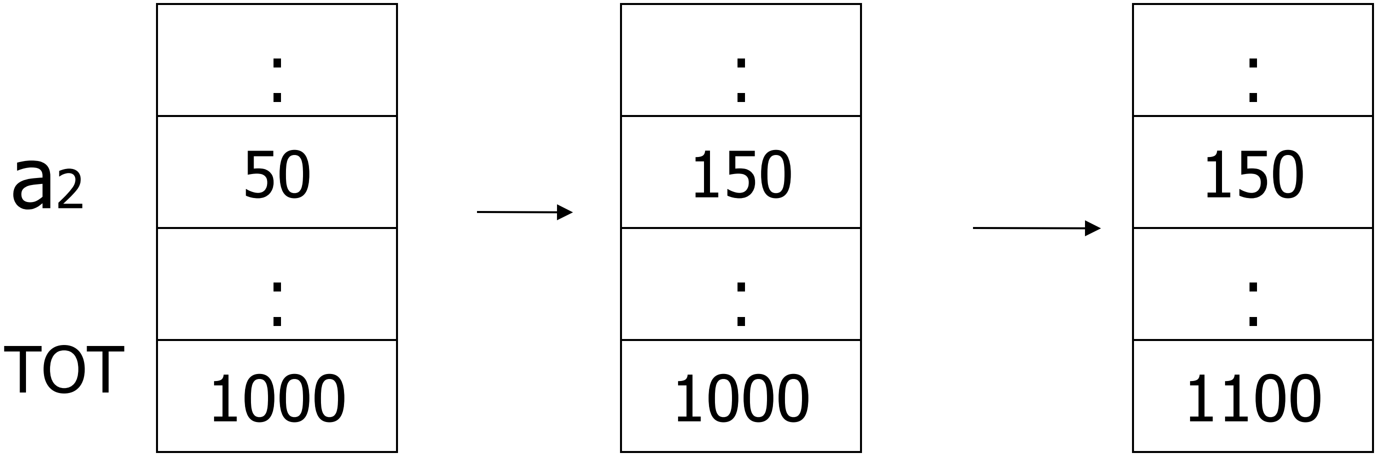
Example: $a_1 + a_2 + \ldots . a_n = TOT$ (constraint)

$$\text{Deposit \$100 in } a_2\text{:} \left\{\begin{array}{l} a_2 \leftarrow a_2 + 100 \\ TOT \leftarrow TOT + 100 \end{array}\right.$$

<u>Example:</u> $a_1 + a_2 + \ldots.\ a_n = TOT$ (constraint)

Deposit \$100 in a_2: $a_2 \leftarrow a_2 + 100$

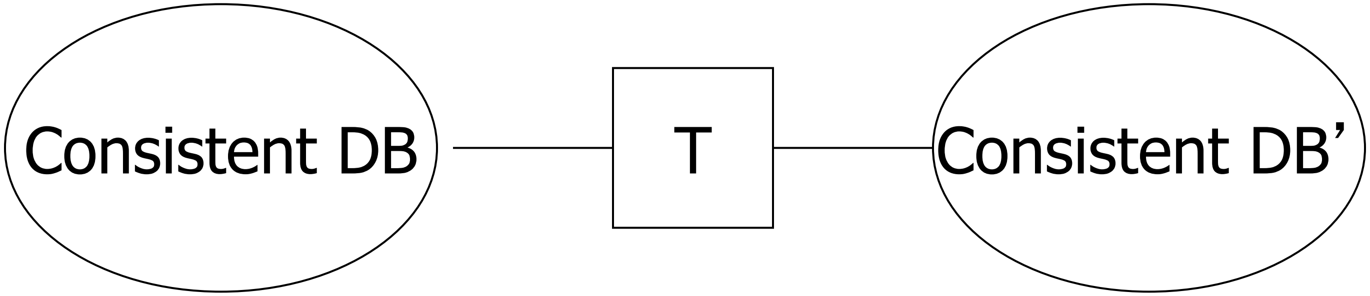
$TOT \leftarrow TOT + 100$



Transactions

- **Transaction**: Sequence of operations executed by one concurrent client that preserve consistency

Transaction: collection of actions that preserve consistency



Big assumption:

If T starts with consistent state +

T executes in isolation

⇒ T leaves consistent state

Correctness (informally)

- If we stop running transactions,
  DB left consistent
- Each transaction sees a consistent DB

Transactions - ACID

- **A**tomicity
  - Either all or no commands of transaction are executed (their changes are persisted in the DB)
- **C**onsistency
  - After transaction DB is consistent (if before consistent)
- **I**solation
  - Transactions are running isolated from each other
- **D**urability
  - Modifications of transactions are never lost

How can constraints be violated?

- Transaction bug
- DBMS bug
- Hardware failure

  e.g., disk crash alters balance of account

- Data sharing

  e.g.: T1: give 10% raise to programmers

  T2: change programmers ⇒ systems analysts

How can we prevent/fix violations?

- Part 13 (Recovery):
  - due to failures
- Part 14 (Concurrency Control):
  - due to data sharing

Will not consider:

- How to write correct transactions
- How to write correct DBMS
- Constraint checking & repair

That is, solutions studied here do not need to know constraints

Data Items:

- **Data Item / Database Object** / …
- Abstraction that will come in handy when talking about concurrency control and recovery
- Data Item could be
  - Table, Row, Page, Attribute value

Operations:

- Input (x): block containing x → memory
- Output (x): block containing x → disk

Operations:

- Input (x): block containing x → memory
- Output (x): block containing x → disk

- Read (x,t): do input(x) if necessary
  t ← value of x in block
- Write (x,t): do input(x) if necessary
  value of x in block ← t

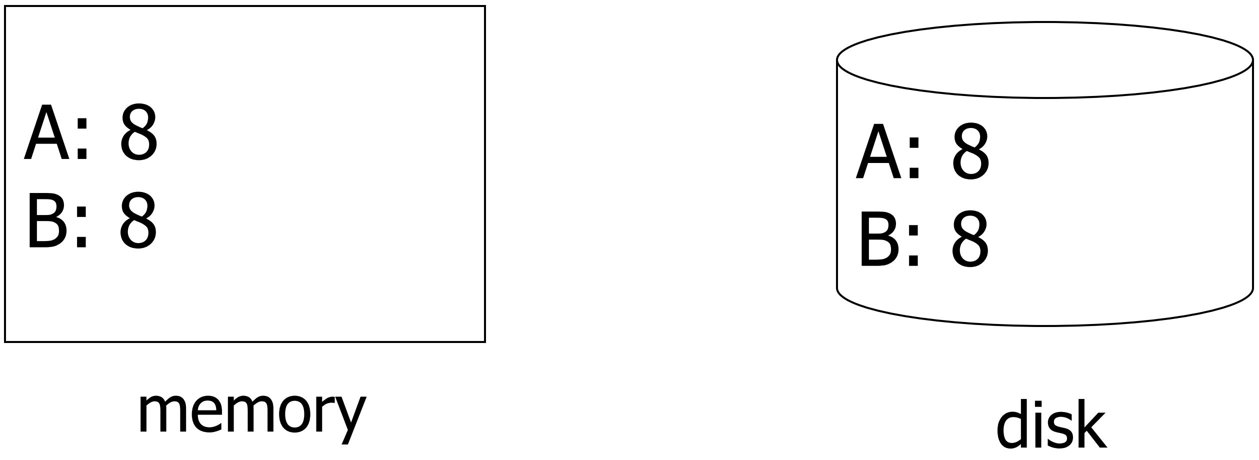
<u>Key problem</u> Unfinished transaction (**Atomicity**)

Example Constraint: A=B

T_1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

```
$T_1$:  Read (A,t);  $t \leftarrow t \times 2$
        Write (A,t);
        Read (B,t);  $t \leftarrow t \times 2$
        Write (B,t);
        Output (A);
        Output (B);
```



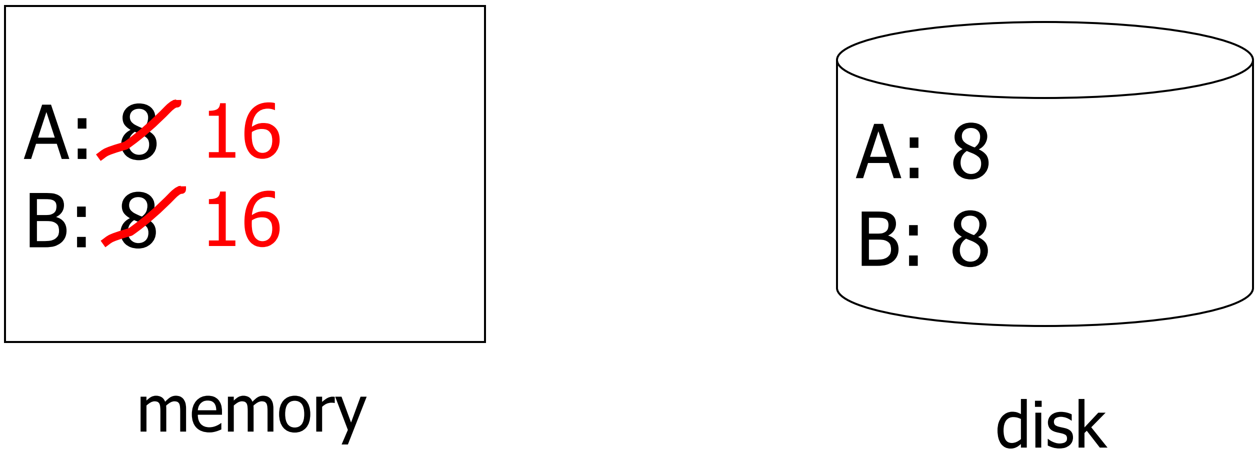
A: 8
B: 8

memory

A: 8
B: 8

disk

T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);


A: ~~8~~ 16
B: ~~8~~ 16

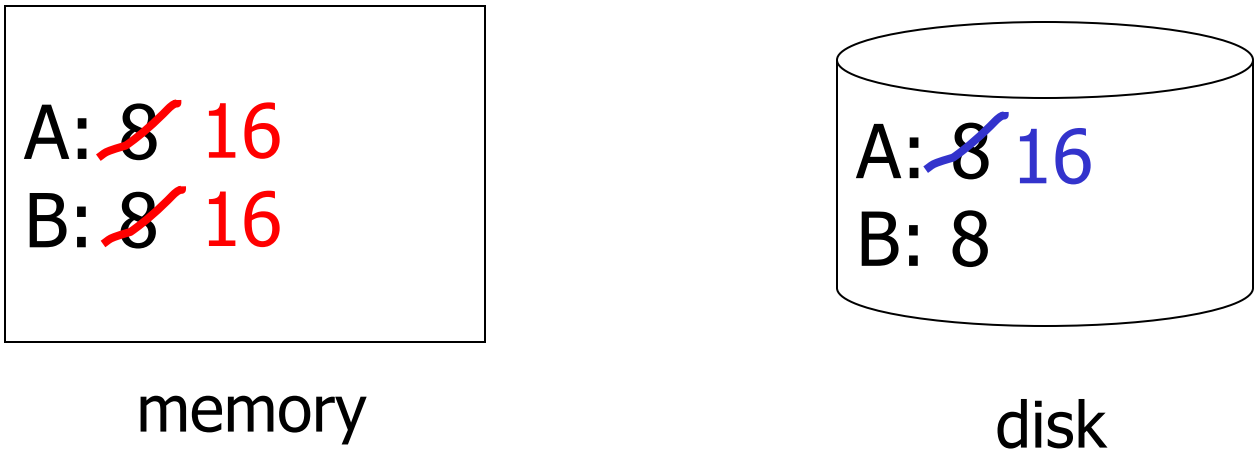
memory

A: 8
B: 8

disk

T_1: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

failure!



A: ~~8~~ 16
B: ~~8~~ 16

memory

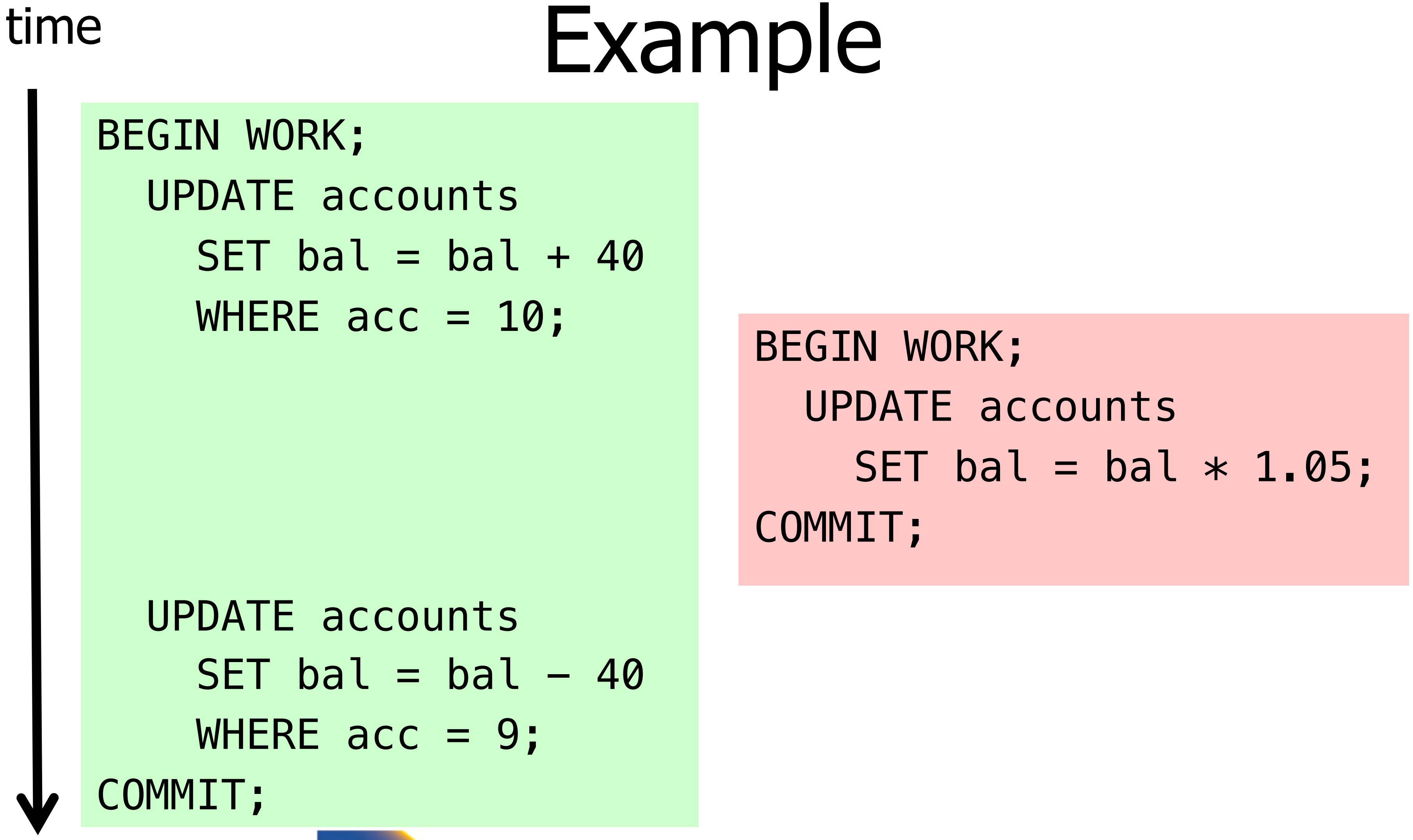
A: ~~8~~ 16
B: 8

disk

Transactions in SQL

- BEGIN WORK
  - Start new transaction
  - Often implicit
- COMMIT
  - Finish and make all modifications of transactions persistent
- ABORT/ROLLBACK
  - Finish and undo all changes of transaction

Example



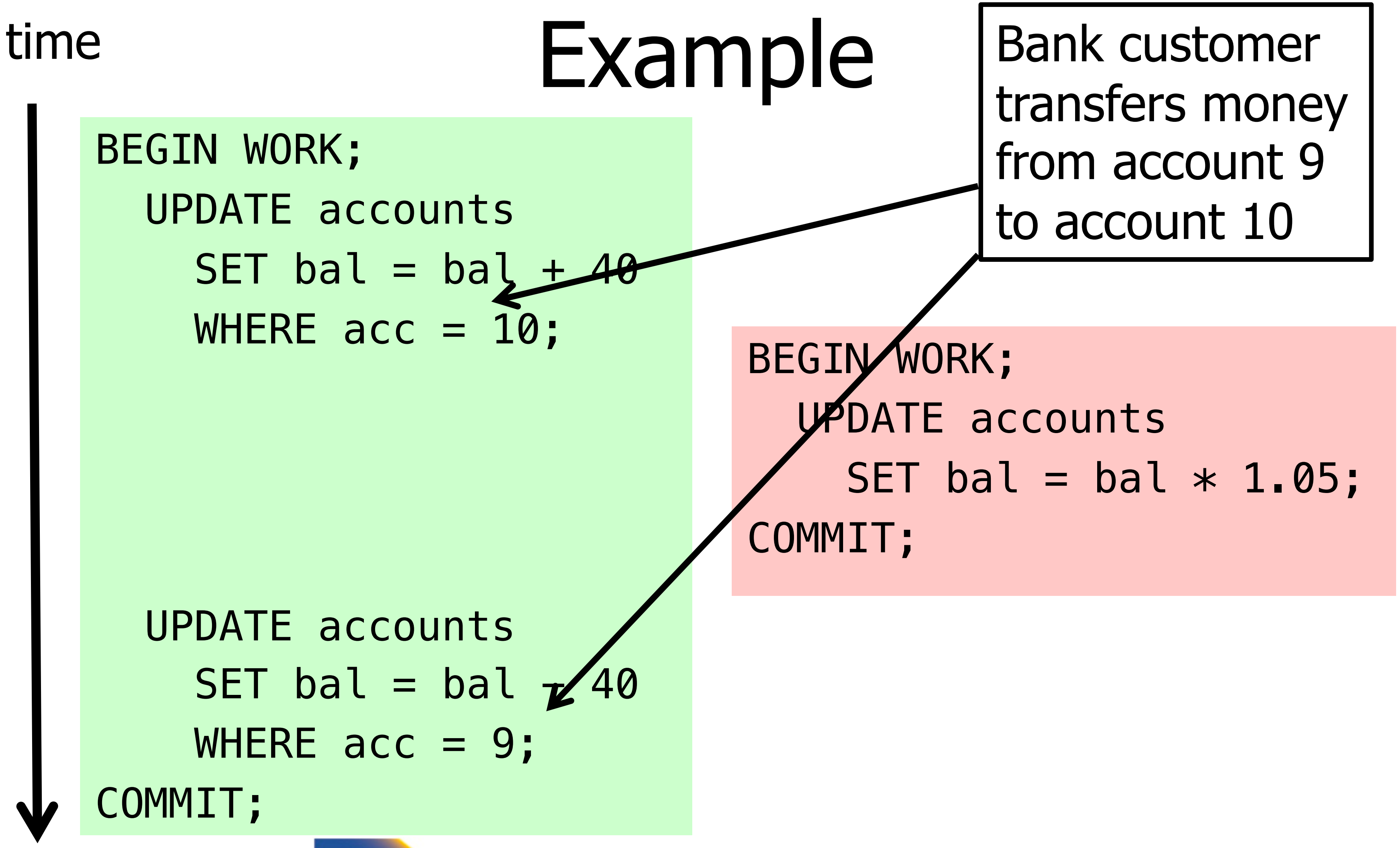
time

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal + 40
    WHERE acc = 10;
```

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal * 1.05;
COMMIT;
```

```
  UPDATE accounts
    SET bal = bal - 40
    WHERE acc = 9;
COMMIT;
```

Example



time

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal + 40
    WHERE acc = 10;




  UPDATE accounts
    SET bal = bal - 40
    WHERE acc = 9;
COMMIT;
```

Bank customer transfers money from account 9 to account 10

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal * 1.05;
COMMIT;
```

Example

time

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal + 40
    WHERE acc = 10;
```

Bank adds interest to all accounts

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal * 1.05;
COMMIT;
```

```
  UPDATE accounts
    SET bal = bal - 40
    WHERE acc = 9;
COMMIT;
```

Potential Problems:

1. Transactions are interrupted
    - No reduction in bal of acc 9
    - Only some accounts got interest
2. Interleaving of Transaction
    - Acc 9 too much interest (before 40 has been deducted)

time

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal
    WHERE acc = 10




  UPDATE accounts
    SET bal = bal – 40
    WHERE acc = 9;
COMMIT;
```

```
    SET bal = bal * 1.05;
COMMIT;
```

Modeling Transactions and their Interleaving

- Transaction is sequence of operations
  - **read**: $\mathbf{r_i(x)}$ = transaction **i** read item **x**
  - **write**: $\mathbf{w_i(x)}$ = transaction **i** wrote item **x**
  - **commit**: $\mathbf{c_i}$ = transaction **i** committed
  - **abort:** $\mathbf{a_i}$ =transaction **i** aborted

$T_1 = r_1(a_{10}), w_1(a_{10}), r_1(a_9), w_1(a_9), c_1$

time

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal + 40
    WHERE acc = 10;





  UPDATE accounts
    SET bal = bal – 40
    WHERE acc = 9;
COMMIT;
```

$T_1 = r_1(a_{10}), w_1(a_{10}), r_1(a_9), w_1(a_9), c_1$

$T_2 = r_2(a_1), w_2(a_1), r_2(a_2), w_2(a_2), r_2(a_9), w_2(a_9), r_2(a_{10}), w_2(a_{10}), c_1$

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal + 40
    WHERE acc = 10;



  UPDATE accounts
    SET bal = bal – 40
    WHERE acc = 9;
COMMIT;
```

Assume we have accounts: a_1, a_2, a_9, a_{10}

```
BEGIN WORK;
  UPDATE accounts
    SET bal = bal * 1.05;
COMMIT;
```

Schedules

- A **schedule S** for a set of transactions $T = \{T_1, \ldots, T_n\}$ is an partial order over operations of T so that
  - **S** contains a prefix of the operations of each T_i
  - Operations of Ti appear in the same order in **S** as in Ti
  - For any two conflicting operations they are ordered

Note

- For simplicity: We often assume that the schedule is a total order

How to model execution order?

- Schedules model the order of the execution for operations of a set of transactions

Conflicting Operations

- Two operations are conflicting if
  - At least one of them is a write
  - Both are accessing the same data item
- Intuition
  - The order of execution for conflicting operations can influence result!

Conflicting Operations

- Examples
  - $w_1(X)$, $r_2(X)$ are conflicting
  - $w_1(X)$, $w_2(Y)$ are not conflicting
  - $r_1(X)$, $r_2(X)$ are not conflicting
  - $w_1(X)$, $w_1(X)$ are not conflicting

Complete Schedules = History

- A **schedule S** for T is complete if it contains all operations from each transaction in T
- We will call complete schedules **histories**

$T_1=r_1(a_{10}),w_1(a_{10}),r_1(a_9),w_1(a_9),c_1$

$T_2=r_2(a_1),w_2(a_1),r_2(a_2),w_2(a_2),r_2(a_9),w_2(a_9),r_2(a_{10}),w_2(a_{10}),c_1$

Complete Schedule

$S=r_2(a_1),r_1(a_{10}),w_2(a_1),r_2(a_2),w_1(a_{10}),w_2(a_2),r_2(a_9),w_2(a_9),$
$r_1(a_9),w_1(a_9),c_1\ r_2(a_{10}),w_2(a_{10}),c_1$

Incomplete Schedule

$S=r_2(a_1),r_1(a_{10}),w_2(a_1),w_1(a_{10})$

Not a Schedule

$S=r_2(a_1),r_1(a_{10}),c_1$

$T_1=r_1(a_{10}),w_1(a_{10}),r_1(a_9),w_1(a_9),c_1$

$T_2=r_2(a_1),w_2(a_1),r_2(a_2),w_2(a_2),r_2(a_9),w_2(a_9),r_2(a_{10}),w_2(a_{10}),c_2$

Conflicting operations

- Conflicting operations $w_1(a_{10})$ and $w_2(a_{10})$
- Order of these operations determines value of a_{10}
- S1 and S2 do not generate the same result

$S_1 = \ldots\ w_2(a_{10})\ \ldots\ w_1(a_{10})$

$S_2 = \ldots\ w_1(a_{10})\ \ldots\ w_2(a_{10})$

Why Schedules?

- Study properties of different execution orders
  - Easy/Possible to recover after failure
  - Isolation
  - -> preserve ACID properties
- Classes of schedules and protocols to guarantee that only "good" schedules are produced