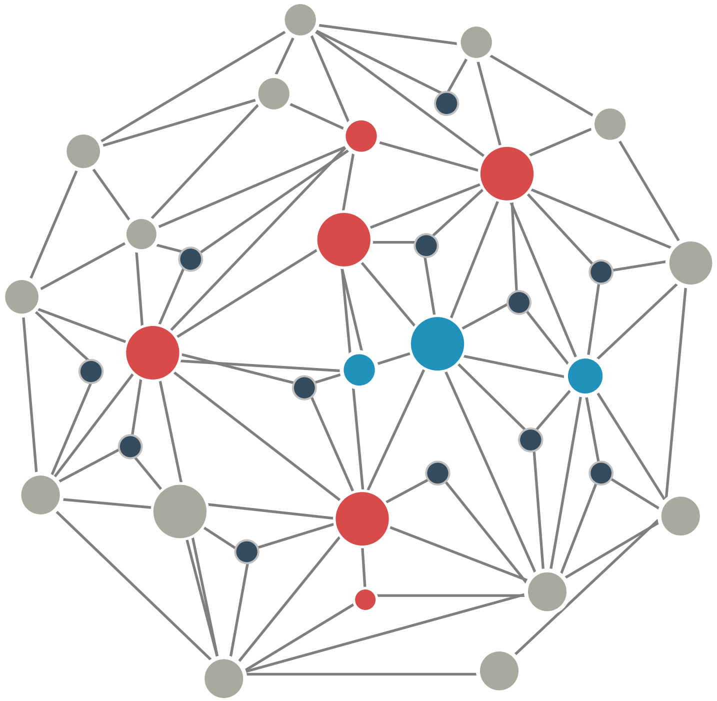
DST2 – Week 13

Transaction and ICA



Zhaoyuan Fang

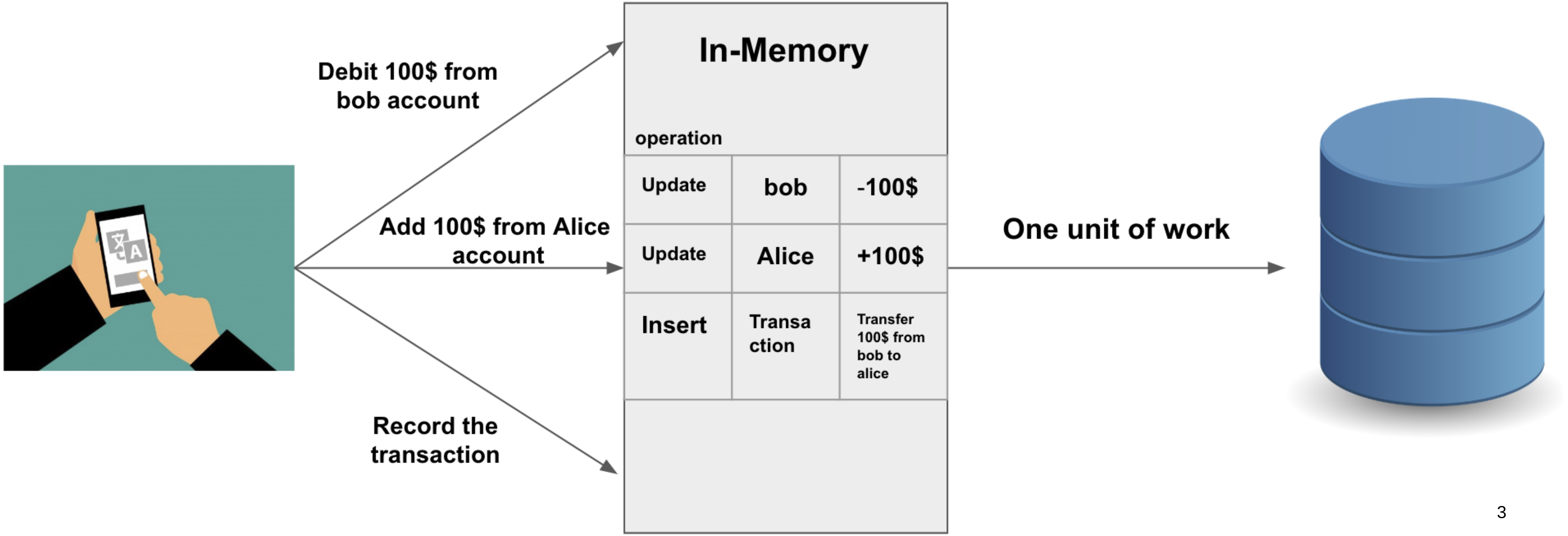
zhaoyuanfang@intl.zju.edu.cn

Week 13 Learning Objectives

- What is database transaction and what are their properties?
- Why transaction isolation
- What is Serializability and conflict Serializability
- What is concurrent transaction
- What are the four read phenomena in concurrent transaction
- ICA annoucement

Transaction

- Suppose you are a banker. A customer Bob comes and this is his requirement: transfer $100 from his account to Alice's account. So you **login** the database system, **operate**, and finally **submit**:



Transaction

- Your “operate – submit” actions form **a unit** of database operations (database transaction):
  - update Bob’s account balance, then update Alice’s account balance.

One transaction

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Bob';
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Alice';
COMMIT;
```

Transaction

- A transaction is **a logical unit**. If you break the operations in a single unit, it might cause problems.
  - What will happen if you only finish one of these two operations?
  - (you finished updates on Bob's balance, but not Alice's.)

One transaction

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Bob';
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Alice';
COMMIT;
```

Transaction

- How many operations can a database transaction have?
- The previous example transaction has two operations.
- But it can have **more or fewer** operations. It may also have only **one** operation.

One transaction

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Bob';
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Alice';
COMMIT;
```

Transaction

- Database **operations** have two types
  - Read(X) operation: query
  - Write(X) operation: insertion, deletion, update
- Then we can distinguish **two types of transactions**:
  - Read-only transaction: only read(X) operations
  - Read-Write transaction: involves write(X) operations
- In other words, **not all transactions update database**
  - Some transactions only retrieve information from a database
  - There is no change in the database state

Transaction

Task :

What is the transaction type in our previous example?

A. Read-only transaction

B. Read-Write transaction

One transaction

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Bob';
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Alice';
COMMIT;
```

Transaction

- A transaction *T* has:
  - A **read-set** (all data items read) and a **write-set** (all data items written)
  - E.g. *T* reads X1 and X2 and updates X3: the read-set is {X1,X2}, write-set is {X3}.

Task :
What is the read-set? What is the write-set? (Hint: specify which tuples)

One transaction

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Bob';
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Alice';
COMMIT;
```

Transaction requirement: database consistency

- Consistent database state: All data integrity constraints are satisfied
  - Constraints from **integrity rules**: e.g. primary key uniqueness, foreign key references, transaction completeness (enforced by DBMS)
  - Constraints from **business rules**: e.g. sum of account balance remains 0 after inter-account money transfer (enforced by programmer)

Transaction requirement: database consistency

- Transactions should maintain database consistency
  - A transaction must begin with a consistent database state, and end with another consistent state
  - But the intermediate state during a transaction could be inconsistent
  - Therefore, improper/incomplete transactions can destroy database consistency (we prefer “all-or-none” execution)

e.g. Transaction (T1) transfers N dollars from X to Y:

```
read_item (X);
X := X – N;
write_item (X);
read_item (Y);
Y := Y + N;
write_item (Y);
```

Transaction requirements – C, A, D

- **Consistency**: On database state, a transaction can only bring it from one consistent state to another by preventing data corruption.
- **Atomicity**: A transaction's operations should be executed as a single "unit" altogether (all-or-none).
- **Durability**: Results of a successful transaction are permanently stored in the system (even in case of power loss or system failures).
- The above is about a single transaction. In reality, multiple transactions can occur at the same time and access the same data items. New problems would come in such cases.

Multiple transactions

- The problem of multiple transactions.
- Consider two transactions T1 and T2:
  - We know, consistency requirement only guarantees the consistency before and after a transaction, but not intermediate states. The intermediate state could be inconsistent.
  - If T1 and T2 are executed one-by-one (serial schedule), then it is safe. Their intermediate data are invisible to each other (**Isolation**).
  - If T1 and T2 are executed in an interleaving mode (nonserial schedule), then it is dangerous. They may access each other’s inconsistent intermediate states (**Isolation violated**)!

Multiple transactions

Serial schedule S1

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

T1/T2 in Isolation

Non-Serial schedule S2

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
	read_item(X);
	$X := X + M$;
write_item(X);	
read_item(Y);	
	write_item(X);
$Y := Y + N$;	
write_item(Y);	

T1/T2 not isolated
may cause inconsistency (T1's update on X overwritten by T2)

Transaction requirements – Now we have ACID!

- **Consistency**: On database state, a transaction can only bring it from one consistent state to another by preventing data corruption.
- **Atomicity**: A transaction’s operations should be executed as a single “unit” altogether (all-or-none).
- **Durability**: Results of a successful transaction are permanently stored in the system (even in case of power loss or system failures).

- **Isolation**: Transactions are executed independently/isolated from each other. Intermediate results of a transaction is not visible to others.

task

Task : **A consistent database state is one in which all _____.**

a. tables have foreign keys b. data integrity constraints are satisfied
c. tables are normalized d. SQL statements only update one table at a time

Task : **_____ means that the data updated during the execution of a transaction cannot be used by other transactions until the first transaction is completed.**

a. Serializability b. Atomicity
c. Isolation d. Time stamping

Solutions: b, c

Transaction requirements – concurrency

- Serial schedules naturally enforce isolation (and consistency), but they also have limitations:
  - Waiting time is long. If T1 is very slow and T2 must wait until T1 is finished, then T2 may keep on waiting.
  - Resource usage is inefficient. Two main resources, CPU and disk I/O. Suppose T1 uses more CPU and T2 uses more I/O. During T1, I/O keeps free; during T2, CPU keeps free. They are just wasting time.
- Non-serial (“concurrent”) schedules allow concurrency:
  - reduce waiting time
  - increase resource usage efficiency
- But: how can non-serial schedules enforce isolation (and consistency)?

Transaction requirements – Serializability

- How can non-serial schedules enforce isolation (and consistency) just like serial schedules?
- Short answer:
  - to make it “equivalent to” a serial schedule.

Transaction requirements – Serializability

Serial schedule S1

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	
	read_item(X);
	$X := X + M$;
	write_item(X);

T1/T2 in Isolation

Non-Serial schedule S3

Time

T_1	T_2
read_item(X);	
$X := X - N$;	
write_item(X);	
	read_item(X);
	$X := X + M$;
	write_item(X);
read_item(Y);	
$Y := Y + N$;	
write_item(Y);	

T1/T2 essentially isolated

Transaction requirements – Serializability

- So the non-serial schedule S3 is actually equivalent to the serial schedule S1.
- We call S3 a “serializable” schedule, and it has the property of serializability.
- Serializable schedule:
  - A schedule S of n transactions that is equivalent to some serial schedule of the same n transactions.
  - It ensures the selected order of concurrent transaction operations creates the same final database state as those produced by serial execution of the transactions.
  - Of course, serial schedules are all serializable.

Serializable schedules and Scheduler

- Scheduler:
  - A special DBMS process that creates a serializable schedule
  - It interleaves the execution of database operations to ensure serializability and isolation of concurrent transactions
  - It bases its actions on concurrent control algorithms (e.g. locking, time stamping) to determine the appropriate order
  - It ensures that two transactions do not update the same data element at the same time to facilitate data isolation

task

Task :

A single-user database system (assuming database access via a single thread) automatically ensures_____ of the database, because only one transaction is executed at a time.

a. serializability and durability

b. atomicity and isolation

c. serializability and isolation

d. atomicity and serializability

Solution: c

Scheduler – task

Task : **The scheduler establishes the order in which the operations within concurrent transactions are executed.**

TRUE or FALSE

Task : **A scheduler facilitates data isolation to ensure that two transactions do not update the same data element at the same time.**

TRUE or FALSE

Solutions: True, True

Example setting – two transactions

Let *T*1 and *T*2 be two transactions that transfer funds from one account to another. Transaction *T*1 transfers \$50 from account *A* to account *B*.

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

Transaction *T*2 transfers 10 percent of the balance from account *A* to account *B*.

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Serial schedules – examples 1 & 2

A=1000, B=2000

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Task:

What's the balance of A and B in each step?

Are the two schedule the equivalent?

Solution:
A=855, B=2145, sum=3000
A=850, B=2150, sum=3000
No

Non-serial/Concurrent schedules – example 1

A=1000, B=2000

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Task:

What's the balance of A and B in each step?

Are the two schedules equivalent?

Solution:
A=855, B=2145, sum=3000
A=855, B=2145, sum=3000
Yes

Non-serial/Concurrent schedules – example 2

A=1000, B=2000

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Task :

What's the balance of A and B in each step?

Are the two schedules equivalent?

Solution:

A=855, B=2145, sum=3000

A=950, B=2100, sum=3050

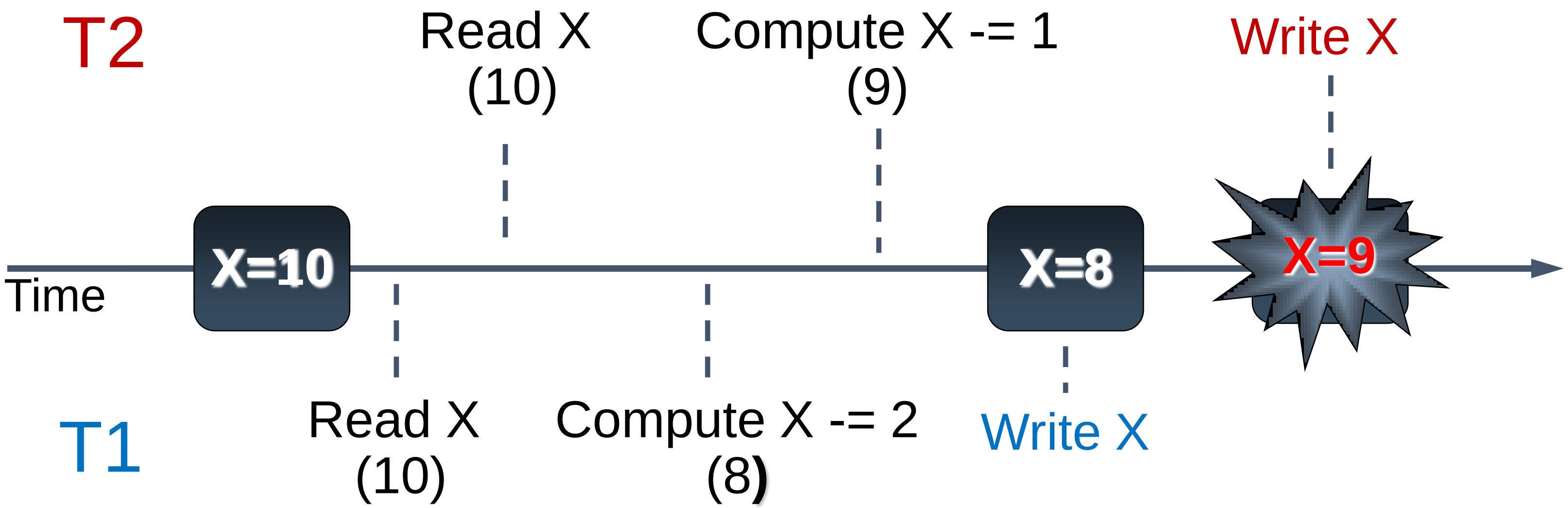
No

A+B not preserved (inconsistency)

Non-serial concurrent transactions: common problems

- **Lost Update (“modified after write”, write-write)**
  - two concurrent transactions update the same data element, and one of the updates is lost (overwritten by the other transaction)
- **Dirty Read (“modified before read”, write-read-rollback)**
  - a transaction reads data from a row that has been modified by another running transaction (but not yet committed)
- **Non-repeatable Read (“modified between two reads”, read-write-read)**
  - a transaction reads the same data element twice, but the data element is changed by another transaction between the two reads
- **Phantom Read (“modified after read”, read-write)**
  - a transaction queries the table, but new rows are added or removed by another transaction to the records being read

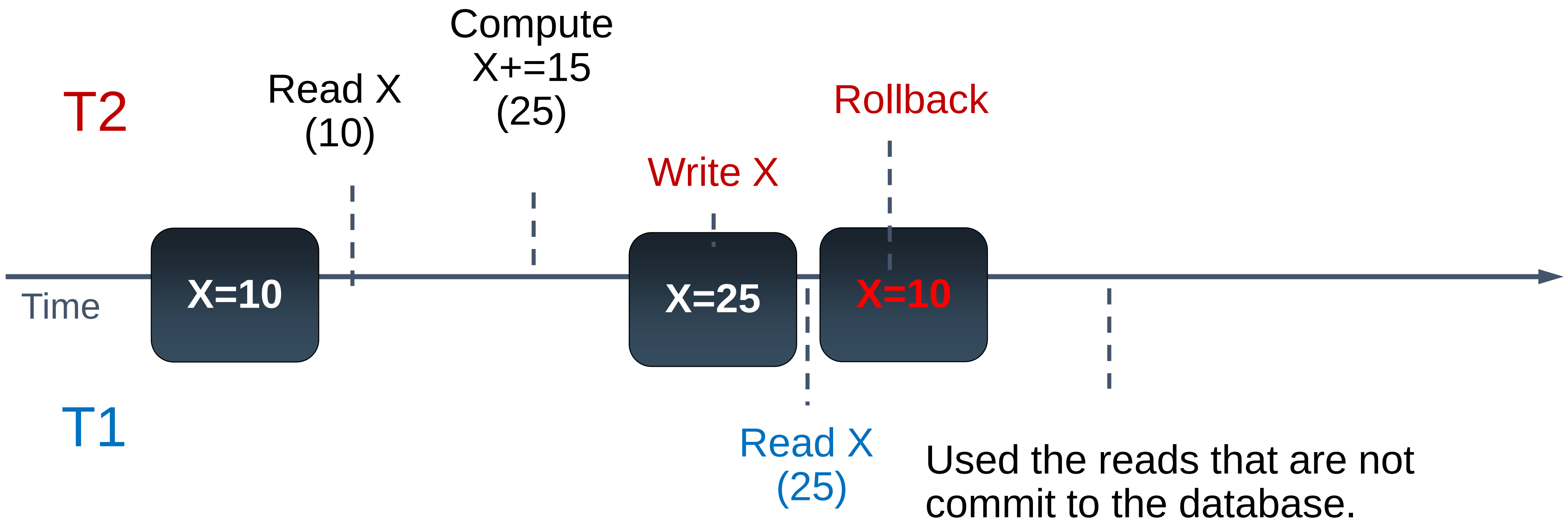
Lost Update



T1 and T2 read the same data and update the data concurrently. The results submit by T2 cause the lost of update by T1.

WRITE - WRITE

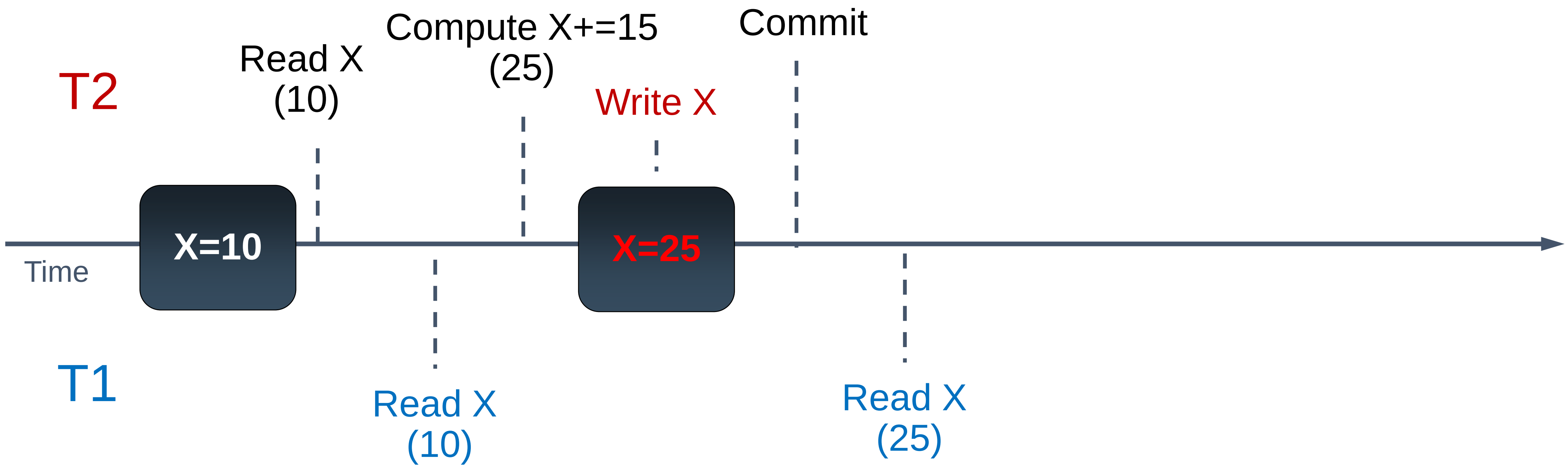
Dirty Read



Just before T1 read some data, T2 update the same data. However, after that, T2 rollback due to some reason. Now the data read by T1 is inconsistent with the data in the database.

WRITE - READ - ROLLBACK

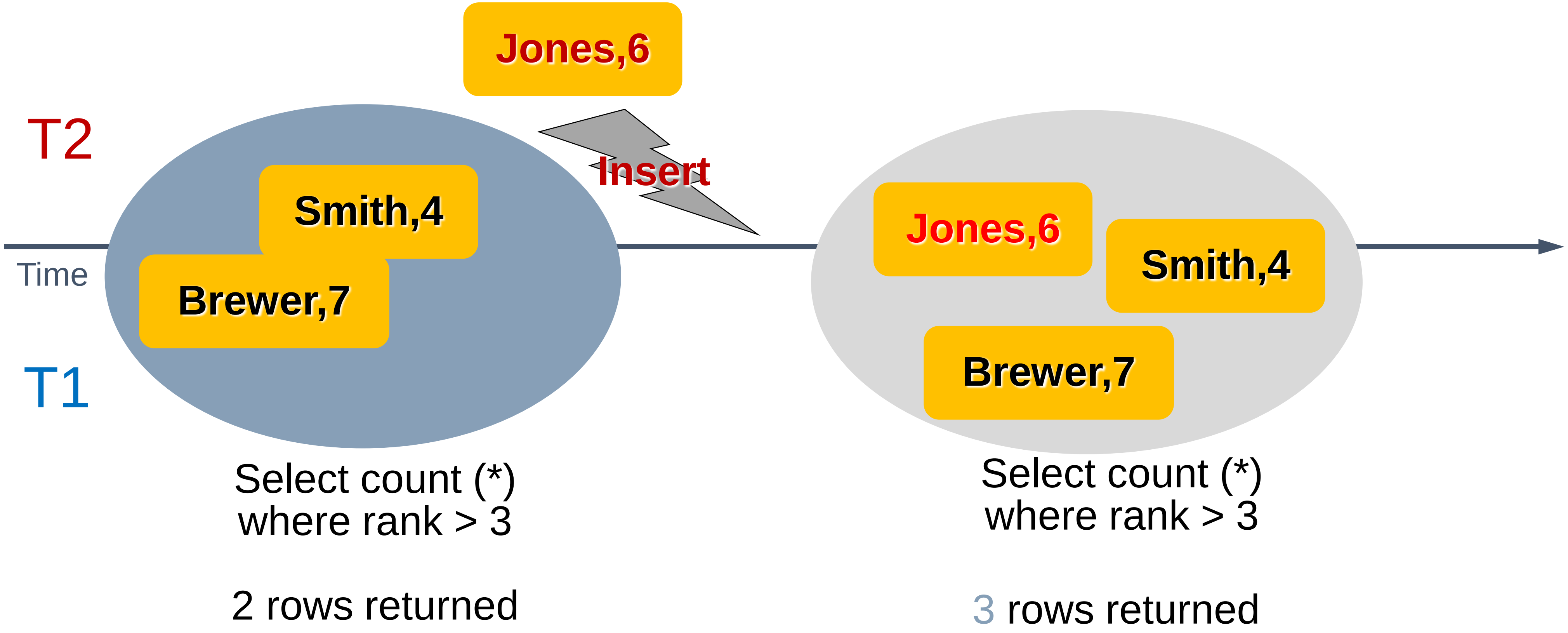
Non-repeatable Read



T1 read some data, T2 then update the data, so when T1 read the data again, the data is inconsistent with previous ones.

READ - WRITE - READ

Phantom Read



T1 read some data based on some conditions, then T2 insert some new data that matches the condition. (If T1 search for data with the same condition, more records are returned.)

READ-WRITE

Task

Task: **One of the three most common data integrity and consistency problems is ______.**

a. lost updates b. disk failures
c. user errors d. deadlocks

Task: **As long as two transactions, T1 and T2, access ______ data, there is no conflict, and the order of execution is irrelevant to the final outcome.**

a. shared b. common
c. unrelated d. locked

Solutions: a, c

Transaction Isolation Levels

- Serializability is strict and may allow too little concurrency.
- Thus there are also some weaker levels of consistency :
  - **Serializable** is the most restrictive one.
  - **Repeatable read** allows only committed data to be read and further requires that, between two reads of a data item by a transaction, no other transaction is allowed to update it.
  - **Read committed** allows only committed data to be read, but does not require repeatable reads. For instance, between two reads of a data item by the transaction, another transaction may have updated the data item and committed.
  - **Read uncommitted** allows uncommitted data to be read. It is the lowest isolation level allowed by SQL.

Isolation levels vs read phenomena

Read phenomena Isolation level	Dirty reads	Lost updates	Non-repeatable reads	Phantoms
Read Uncommitted	may occur	may occur	may occur	may occur
Read Committed	don't occur	may occur	may occur	may occur
Repeatable Read	don't occur	don't occur	don't occur	may occur
Serializable	don't occur	don't occur	don't occur	don't occur

Task 17 : In different isolation levels, would those read phenomena occur or not?

For example, if we want to know when Lost updates does not occur, then from the above table, we can find it should be in isolation levels: Serializable, or Repeatable Read

Serializable schedule - How to determine it more technically?

- Given a schedule, how can we determine if it is a serializable schedule? By definition:
  - interleaved execution of transactions that are **equivalent** to some serial schedule of these transactions
- We need to define schedule equivalence
  - One way is: two schedules that yield the same results on the same transactions
  - But two schedules may yield same results accidently, i.e. simply by chance
  - So we need some better criteria: conflict equivalence

Conflict equivalence – example

- First, we can simplify operations in transactions and consider only two key operations: read and write

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Conflict equivalence – example

- Now, let us consider a schedule *S.* There are two consecutive operations, *I* and *J* , of transactions *Ti* and *Tj* , respectively ($i \neq j$).
- If *I* and *J* refer to different data items, then we can **swap** *I* and *J* without affecting the results of any instruction in the schedule.
- However, if *I* and *J* refer to the same data item *Q*, can we **swap** them?

1. I = read(Q), J = read(Q);
2. I = read(Q), J = write(Q);
3. I = write(Q), J = read(Q);
4. I = write(Q), J = write(Q);

Task: Would case 1 to case 4 have different results if we swap the order?

So if two operations on the *same data* has at least one "*write*" in them, then they **can not swap**, otherwise will cause a conflict (loss of equivalence) before and after swapping. We say I and J operations are conflict in above case 2~4.

Conflict equivalence – definition

- If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S' are **conflict equivalent**.
- Or more simply, S and S’ are **equivalent**.

S1

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

S2

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

S3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Task:

1. Is S1 and S2 equivalent?
2. And S2 vs S3?

No conflicts for A and B are different data items.

Serializable schedule – how to determine it

- In this example, S1 is equivalent to S2, and S2 is equivalent to S3, so S1 and S3 are also equivalent. Notice that S3 is a serial schedule and S1 is equivalent to it => S1 is a **(conflict) serializable schedule**.
- In general, schedule *S* is **(conflict) serializable** if it is (conflict) equivalent to a serial schedule. This is exactly what we have already introduced intuitively.

S1

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

S2

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

S3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Serializable schedule – determine it with precedence graph

- We now present a simple and efficient method for determining the conflict serializability of a schedule.
- Consider a directed graph, the **precedence graph**, from *S*.
  - Vertices: all the transactions in the schedule. One transaction, one node
  - Edges: an edge $T_i \rightarrow T_j$ is formed if **any** of the three conditions holds
    - write(*Q*) of *Ti* ***before*** read(*Q*) of *Tj*
    - read(*Q*) of *Ti* ***before*** write(*Q*) of *Tj*
    - write(*Q*) of *Ti* ***before*** write(*Q*) of *Tj*

Serializable schedule – determine it with precedence graph

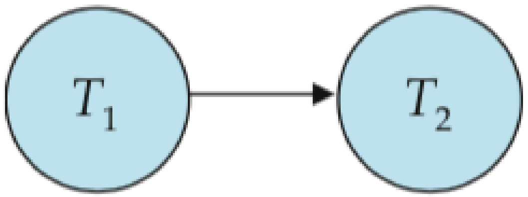
S1

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Simplified table

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Precedence graph



conflict serializable

Serializable schedule – determine it with precedence graph

Task 12 :

1. What are the simplified table and precedence graph?
2. conflict serializable or not?

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Task solution

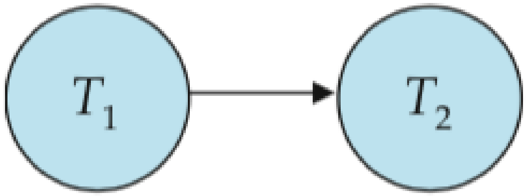
Task 12 :
1. What are the simplified table and precedence graph?
2. conflict serializable or not?

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B) commit	
	read(B) $B := B + temp$ write(B) commit

Simplified table

T_1	T_2
read(A) write(A)	
	read(A) write(A)
read(B) write(B)	
	read(B) write(B)

Precedence graph



Serializable schedule – determine it with precedence graph

Task 13 :

1. What are the simplified table and precedence graph?
2. conflict serializable or not?

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

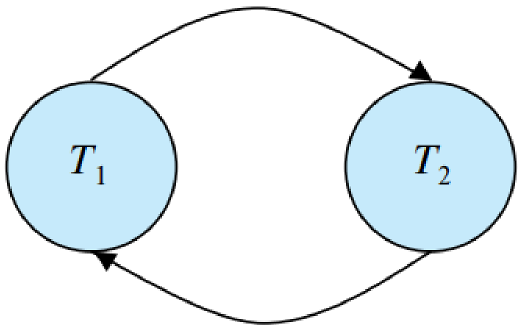
Task solution

Task 13 :

1. What are the simplified table and precedence graph?
2. conflict serializable or not?

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

T1	T2
read(A)	
	read(A)
	write(A)
	read(B)
write(A)	
read(B)	
write(B)	
	write(B)



Serializable schedule – determine it with precedence graph

- If there is Ti -> Tj in the **precedence graph**, then in any equivalent schedule S', Ti need to be before Tj
- If the precedence graph for S has a cycle, then Ti is both before and after Tj in any equivalent schedule S', and no serial schedules possibly have this, thus schedule S is not conflict serializable.
- If the graph contains no cycles, then the schedule S is conflict serializable.

Task 14 : Is this schedule conflict serializable?

How about the output of this schedule vs serial schedule? Are the output the same?

Not serializable (has a cycle in precedence graph).

It is possible to have two schedules that produce the same outcome, but that are not conflict equivalent.

T_1	T_5
read(A) $A := A - 50$ write(A)	
	read(B) $B := B - 10$ write(B)
read(B) $B := B + 50$ write(B)	
	read(A) $A := A + 10$ write(A)

Transaction Atomicity and Durability

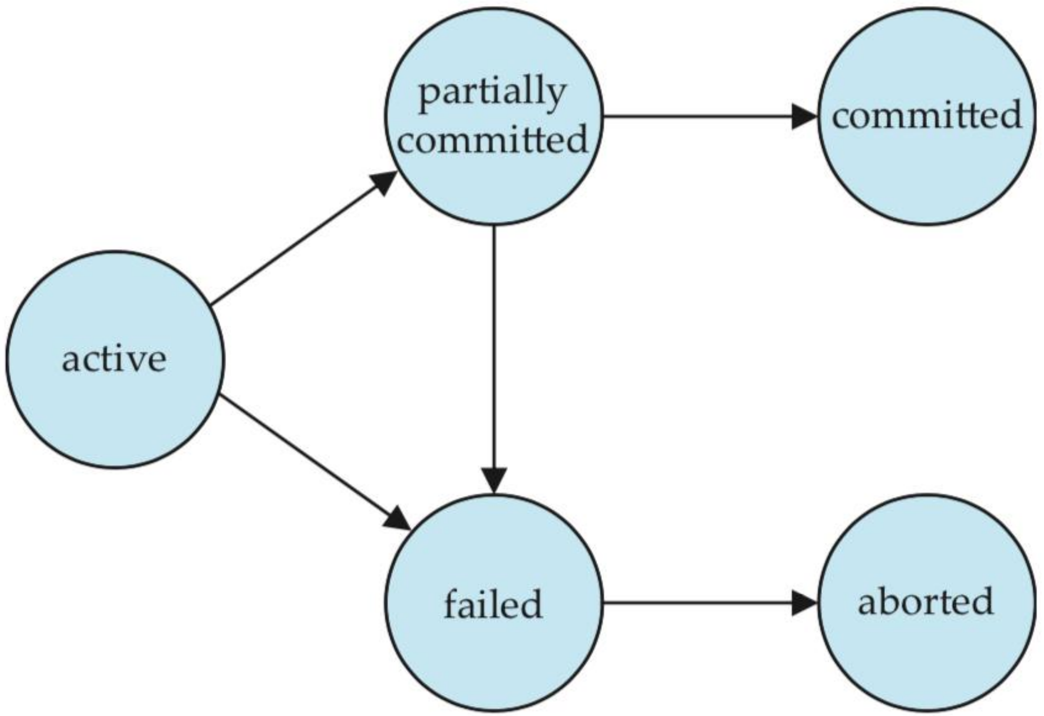
- Transaction in “all-or-none” execution mode (**Atomicity**):
  - A transaction initiates and can complete its execution successfully (**committed**)
  - But it may also fail at somewhere and can not complete successfully (**aborted**)
  - An aborted transaction will cause all its executions undone (**rolled back**)

- Once a transaction has been committed:
  - we cannot undo its effects by aborting it (**Durability**).

Transaction states in a diagram

A transaction must be in one of these states:

- **Active**, the initial state; the transaction begins from here.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can not proceed.
- **Aborted**, the transaction rolled back and restored to previous state.
- **Committed**, after successful completion.



Transaction Management with SQL

- COMMIT and ROLLBACK is widely supported in SQL implementations
  - COMMIT: marks the end of transaction and all changes are permanently recorded within the database
  - ROLLBACK: marks the abortion of a transaction and all changes of the transaction are reversed. The database is rolled back to its previous consistent state.
- The begin of a transaction is different for different DBMSs
  - MySQL: START TRANSACTION (BEGIN);
  - PostgreSQL: BEGIN;
  - SQL server: BEGIN TRANSACTION;
  - Oracle: SET TRANSACTION;

Transaction management in MySQL

- By default, MySQL runs with autocommit mode enabled
  - Each statement is a transaction, as if it were surrounded by BEGIN and COMMIT
  - If an error occurs during statement execution, the statement is automatically rolled back (you can not control it using ROLLBACK)
- To switch to transaction mode (supports multiple statements/operations), use the TRANSACTION statements:
  - BEGIN; …; COMMIT;
  - With BEGIN, autocommit is disabled until the transaction ends with COMMIT or ROLLBACK

BEGIN-ROLLBACK-COMMIT

DEMO:

- *Open your MySQL:*
- *choose any database for a test*

```
CREATE TABLE accounts
(id serial,
name varchar(30) NOT NULL,
balance numeric(15,2) NOT NULL,
PRIMARY KEY(id));
```

- *If we insert directly, we have no control over transaction*

```
INSERT INTO accounts(name,balance)
VALUES('Bob',10000);
```

```
+----+-------+----------+
| id | name | balance  |
+----+-------+----------+
|  1 | Bob  | 10000.00 |
+----+-------+----------+
```

BEGIN-ROLLBACK-COMMIT

DEMO:

- *If we use TRANSACTION:*

```
BEGIN;
INSERT INTO accounts(name,balance)
  VALUES('Alice',12);
ROLLBACK;
SELECT * FROM accounts;
```

```
+----+-------+-----------+
| id | name | balance  |
+----+-------+-----------+
|  1 | Bob  | 10000.00 |
+----+-------+-----------+
```

```
BEGIN;
INSERT INTO accounts(name,balance)
  VALUES('Alice',12);
COMMIT;
SELECT * FROM accounts;
```

```
+----+--------+-----------+
| id | name   | balance  |
+----+--------+-----------+
|  1 | Bob    | 10000.00 |
|  2 | Alice |    12.00    |
+----+--------+-----------+
```

BEGIN-COMMIT

DEMO:

- *Suppose we want to update the balance of Bob from 10000 to 10:*

```
BEGIN;
UPDATE accounts
SET balance = balance - 9990
WHERE id = 1;
COMMIT;

SELECT * FROM accounts;
```

```
+----+--------+-----------+
| id | name   | balance   |
+----+--------+-----------+
|  1 | Bob    |    10.00  |
|  2 | Alice  |    12.00  |
+----+--------+-----------+
```

BEGIN-COMMIT

Task: transfer 3 USD from Bob's account to Alice's account with transaction. (using UPDATE, SET)

```
BEGIN;
UPDATE accounts
    SET balance = balance - 3
    WHERE id = 1;
UPDATE accounts
    SET balance = balance + 3
    WHERE id = 2;
COMMIT;

SELECT * FROM accounts;
```

Summary

- Transaction
  - ACID properties
    - Atomicity
    - Consistency
    - Isolation
    - Durability
- Transaction state
  - Active
  - Partially committed
  - Failed
  - Aborted
  - Committed
- Schedules, Scheduler
- Concurrent transaction
- Read phenomena in concurrent transaction
  - Lost update
  - Non-repeatable read
  - Phantom read
  - Dirty read
- Serializability
- Conflict serializability
- Precedence graph
- Transaction isolation level
  - Read uncommitted
  - Read committed
  - Repeatable read
  - Serializable

Database roadmap

Week1: “database newbie”
- Database history
- Relational databases
- Relational algebra operations
- Null values

Week2: “SQL data hunter”
- SELECT, AND, AVG, BETWEEN, COUNT, DISTINCT, EXISTS, FROM, GROUP BY , LIMIT, HAVING IN, IS NULL, LIKE, MAX, MIN, NOT, OR, ORDER BY, Subquery, SUM, WHERE, wildcard

Week3: “Advanced SQLer”
- SQL datatype, schemas
- CREATE table, INSERT data
- Integrity Constraints
- Function, Procedural Constructs
- Embedded SQL

Week4: “Database designer”
- E-R relationship, ERD
- Attributes and its variation
- Cardinality and its different forms
- Unary, Binary, Ternary relationship

Week5: “SQL master”
- Database normalization
- 1NF, 2NF, 3NF
- Database index

Week6: “database transactor”
- Transaction management
- ACID, Schedules
- Concurrent transaction
- Serializability

Thanks for finish up the journey with me ‘DATABASE ADVENTURER’

Announcement of ICA - mini-project

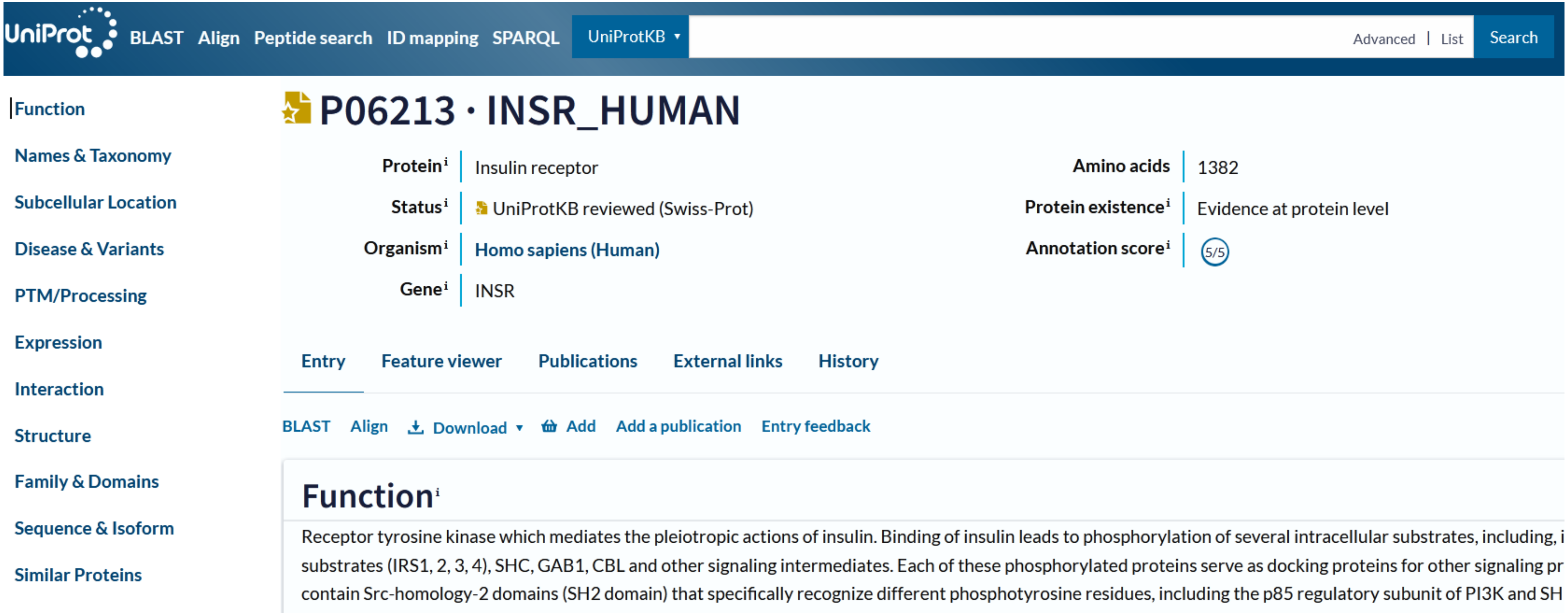
- Mini-project: build a mini-database for human genomic information.
  - The deadline for this ICA is **11:00am of the Monday of Feb 13th 2023**.
  - You will receive your provisional marks and feedback for this ICA on or before 11:00am of Mar 6th 2023.
  - This ICA uses Blackboard and requires individual work for problem solving.

Announcement of ICA - mini-project

Data source: **Human** protein information from UniProt (http://www.uniprot.org/)

Choose **any topic(s)** of interest to yourself: protein family? organs? disease? Check some references.

Collect **data** for your topic: UniProt has rich data, you can choose some from it



Announcement of ICA - mini-project

Instructions:

- What is the database for? Do not be too trivial and think over some interesting topic(s)
- Draw an ER diagram to design the database, with 5~10 tables (neither too simple nor too complex), make sure it is in 3NF and prove it is truly in 3NF
- Include primary key for each table, and use foreign keys to relate tables if needed
- Collect data & insert data into tables (do NOT be too huge, only 10s-100s tuples are sufficient to demonstrate your database)
- Show by Use Cases: demonstrate the usage of your database, with SQL codes and results
- Dump the database as a SQL file: mysqldump -u root -p --databases DATABASE_NAME > **my_database.sql**

Announcement of ICA - mini-project

Required files in your final uploading:

- A **database SQL file** (<30MB): **my_database.sql**
- A **PDF file** (<= 6 pages) of database documentation: ER diagram, description (why create this, what used for, 3NF, optional references < 5), use cases (i.e. describing usage examples, you should also include SQL commands here)
- A **Use Case SQL file**: SQL commands for use cases in a file **use_cases.sql**, so that I can copy-and-paste easily to check it in my MySQL. Add some explanations in your code (e.g. %* *use case 1: show blablabla* *%).

Zip the three files above into a single zip file:

- **YourRollNumber_YourName.zip** (e.g. 10000_BobSmith.zip, the ZIP file size < 20MB)
- Make sure it is ZIP. Before submission, test if you can decompress it successfully.

Announcement of ICA – Marking criteria

Range	Database design	Documentation	Use Case SQL
90-100%	Successful importing into MySQL, very good design and rich information	Very pretty ERD, very good rationale and description of database & usages	Successful running, nice SQL techniques, very good examples & annotations
80-90%	Successful importing, good design, good information	Pretty ERD, good rationale and documentation of database & usages	Successful running, good SQL techniques, good annotations
70-80%	Successful importing, plain design & information	clear ERD, plain rationale, clear documentation of database & usages	Successful running, plain SQL, simple annotations
60-70%	Fail importing, but seems OK in content	problematic ERD, plain or poor rationale, unclear documentation	Fail to run, SQL good or plain, annotations good or ok
30-59%	Fail importing and poor content	poor ERD, poor rationale, unclear documentation	Fail to run, and with poor SQL and annotations
0-29%	Very poor	Very bad ERD, poor rationale & documentation	Fail to run, and with very poor SQL and annotations

Thank You!