Australian National University

Database Transactions – Part 1

Introduction

Transaction – A Classical Example

- **Scenario:** Suppose that Steve's account balance is **$1000** and Bob's balance is **$200**. Now Steve wants to transfer **$500** into Bob's account.
- There are several steps involved in transferring the money:
  1. **Check** Steve's balance;
  2. **Update** Steve's balance;
  3. **Check** Bob's balance;
  4. **Update** Bob's balance.
- Steve later checked his balance (it was **$500**), which looked good to Steve. However, Bob told Steve that he hadn't received his money yet (still **$200** in Bob's account instead of **$700**).

  Question: What did happen?

Transaction – A Classical Example

- **Reason:** Due to power outage, the system **stopped working just after updating** Steve's balance.
- **Task:** Transfer **$500** from Steve's account to Bob's account

1.
```
SELECT balance FROM ACCOUNT
WHERE name = 'Steve';
```
2.
```
UPDATE ACCOUNT
SET balance = balance-500
WHERE name='Steve';
```
3.
```
SELECT balance FROM ACCOUNT
WHERE name = 'Bob';
```
4.
```
UPDATE ACCOUNT
SET balance = balance+500
WHERE name = 'Bob';
```

Operations	Steve	Bob
before 1	$1000	$200
after 1	$1000	$200
after 2	$500	$200
after 3	$500	$200
after 4	$500	$700

Transaction – A Classical Example

- We need an approach to ensure that
  - either the balances of Steve and Bob remain unchanged **if the money transfer fails**
  - or Steve's balance is **$500** and Bob's is **$700** **if the money transfer succeeds.**

1.
```
SELECT balance FROM ACCOUNT
WHERE name = 'Steve';
```

2.
```
UPDATE ACCOUNT
SET balance = balance-500
WHERE name='Steve';
```

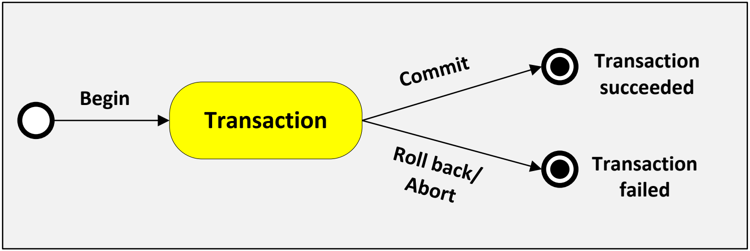
3.
```
SELECT balance FROM ACCOUNT
WHERE name = 'Bob';
```

4.
```
UPDATE ACCOUNT
SET balance = balance+500
```

Operations	Steve	Bob
before 1	$1000	$200
after 1	$1000	$200
after 2	$500	$200
after 3	$500	$200
after 4	$500	$700

What is a Transaction?

- DBMSs provide **transaction support** for solving this kind of problem.
- A **transaction** is **a sequence of database operations grouped together** for execution as **a logic unit** in a DBMS.
    - Different from an execution of a program outside the DBMS (e.g., a C program) in many ways!



What is a Transaction?

- Database applications often access a database **by transactions** rather than individual operations.
  - e.g., large databases and hundreds of concurrent users: banking, supermarket checkout, airline reservation, online purchasing, etc.
- **Why transactions?** They can **enforce data integrity** in the following situations:
  - multiple users may modify and share data at the same time;
  - transaction, system, and media failures may happen from time to time.
- **What does a transaction look like?**
  - `INSERT`, `SELECT`, `UPDATE`, `DELETE`, `BEGIN`, `COMMIT`, `ABORT` (`ROLLBACK`), etc. from a high-level language perspective;
  - `read`, `write`, `begin`, `commit`, `abort` at the internal process level.

Transaction – Language Level

- **Database operations** of a transaction (at the SQL language level) may include: SELECT, INSERT, UPDATE, DELETE.
- **Other operations**: BEGIN, COMMIT, ABORT (ROLLBACK)

```
BEGIN TRANSACTION
```

1.
```
SELECT balance FROM ACCOUNT WHERE name = 'Steve';
```

2.
```
UPDATE ACCOUNT
SET balance = balance-500 WHERE name='Steve';
```

3.
```
SELECT balance FROM ACCOUNT WHERE name = 'Bob';
```

4.
```
UPDATE ACCOUNT
SET balance = balance+500 WHERE name = 'Bob';
```

```
COMMIT
```

Transactions - Internal Process Level

- **Basic operations** of a transaction (at the internal process level) are
  - `read(X)`: loads object X into main memory;
  - `write(X)`: modifies in-memory copy of object X (and writes it to disk later on);
- **Granularity of objects**: tables, rows, cells, or memory pages,
- **Other operations:**
  - `begin`: marks the beginning of a transaction;
  - `commit`: signals a successful end of the transaction - all changes can safely be applied to the database permanently;
  - `abort`: signals the transaction has ended unsuccessfully - undo all operations of the transaction.

Transactions - Internal Process Level

```
T: BEGIN TRANSACTION
T: SELECT balance FROM ACCOUNT WHERE name = 'Steve';
T: UPDATE ACCOUNT SET balance = balance-500 WHERE name='Steve';
T: SELECT balance FROM ACCOUNT WHERE name = 'Bob';
T: UPDATE ACCOUNT SET balance = balance+500 WHERE name = 'Bob';
T: COMMIT;
```

Objects:

- A - Steve's account balance;
- B - Bob's account balance.

Steps	*T*
1	read(A)
2	write(A) (A:=A-500)
3	read(B)
4	write(B) (B:=B+500)
5	commit

Database Transactions – Part 2

ACID Properties

ACID Properties

- DBMSs ensure the following properties of transactions.
  - **Atomicity**:
    - The execution of each transaction is atomic, i.e., **either all operations are completed or not done at all**.
  - **Consistency**:
    - The states of a database are consistent (w.r.t. defined business rules) **before and after each transaction**.
  - **Isolation**:
    - Execution results of each transaction should be **unaffected by other concurrent executing transactions.**
  - **Durability**:
    - Once a transaction has been successfully completed, **its effects should persist in the database**.

Note: These properties are not independent from one another, but **atomicity is the central property**.

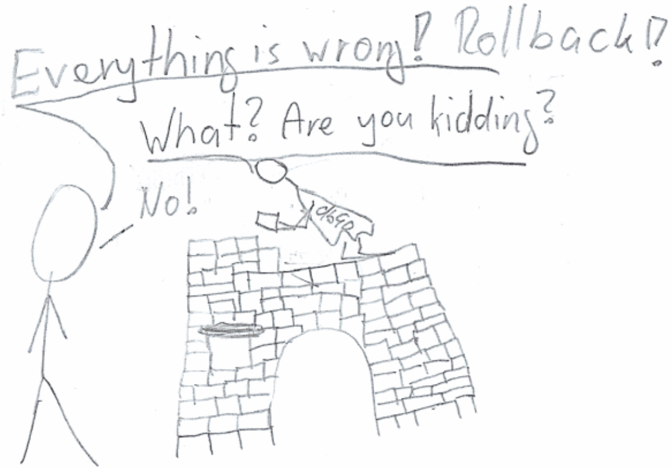
Atomicity

- **Atomicity** requires that we execute a transaction to completion with only **two possibilities**:
  - **ALL**: all the operations are executed;
  - **NONE**: none of the operations are executed.
- If a transaction fails to complete for some reason, it may leave database in an inconsistent state. Thus a DBMS **must remove effects of partial transactions** to ensure atomicity.

  Example: The money can only be taken from Steve's account if the money has been transferred into Bob's account.

Operations	Steve	Bob	
before 1	$1000	$200	**None are executed.**
after 1	$1000	$200	
after 2	$500	$200	
after 3	$500	$200	
after 4	$500	$700	**All are executed.**

Atomicity



Consistency

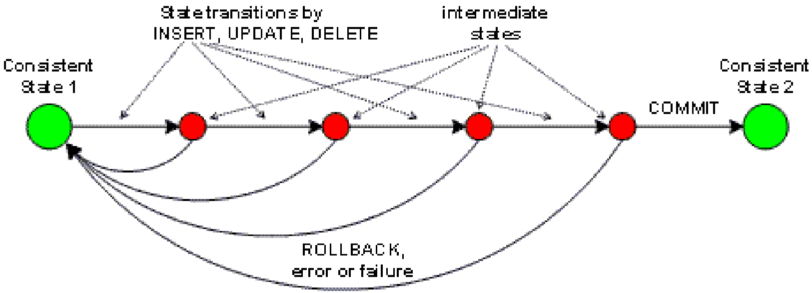
- **Consistency** requires that, each transaction should **preserve the consistency of the database**.
- **Note:** Intermediate states may be inconsistent.

  Example: Suppose that we have

  Steve's account balance + Bob's account balance = $1200,

Operations	Steve	Bob	
before 1	$1000	$200	$1000+$200=$1200
after 1	$1000	$200	
after 2	$500	$200	**Not required to be consistent.**
after 3	$500	$200	
after 4	$500	$700	$500+$700=$1200

Consistency [1]



- The database is in a consistent state before and after executing the transaction, but is not necessarily consistent in intermediate states.

[1] The figure is taken from http://maxdb.sap.com

Isolation

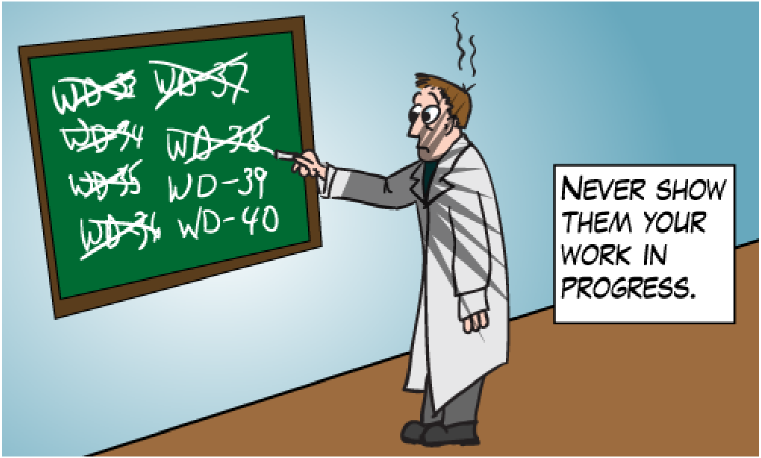
- **Isolation** requires that transactions are **isolated from one another**.

  Example: Other transactions can't see the changes on objects *A* (Steve's account balance) and *B* (Bob's account balance) until the transaction for the money transfer is completed.

T_1
read(A)
write(A) (A:=A-500)
read(B)
write(B) (B:=B+500)
commit

T_2
read(A)
write(A) (A:=A+400)
commit

Isolation [2]



[2] The figure is taken from http://michaeljswart.com/

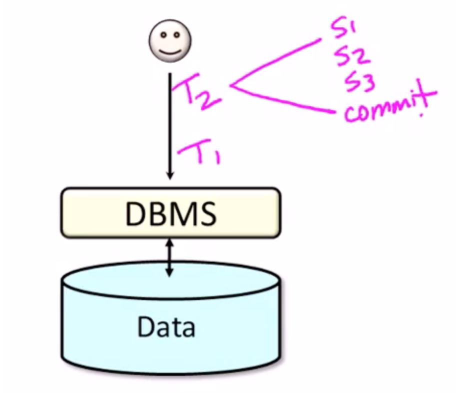
Durability

- **Durability** requires that once the transaction is successfully completed, its changes to the database **must be persistent despite failures**.
- The decision is irrevocable: once committed, the transaction cannot revert to abort. **Changes are durable**.
- **Example**: Once Steve received the notification:

  "**$500 has been successfully transferred to Bob's account**",

  the money can't go back to Steve's account and must appear in Bob's account.

Durability [3]



[3] The figure is taken from http://toyhouse.cc/profiles/blogs/the-acid-properties-of-transactions

Australian National University

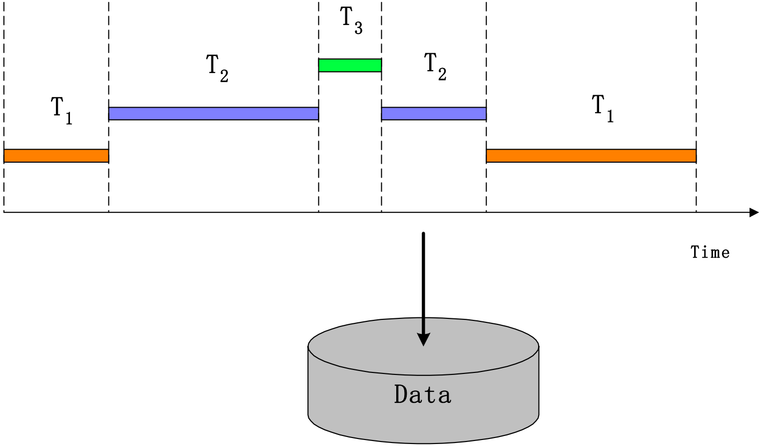
Database Transactions – Part 3

Concurrent Transactions

Australian National University

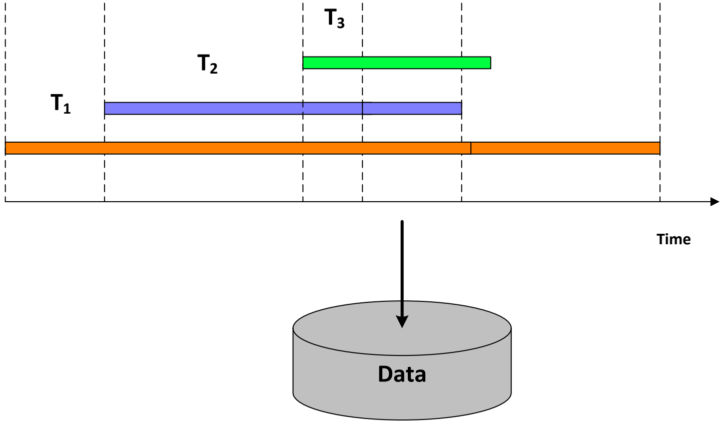
Concurrent Transactions

- **Interleaved processing**: transactions are interleaved in a single CPU.



Concurrent Transactions

- **Parallel processing**: transactions are executed in parallel in multiple CPUs.



Concurrent Transactions

- Executing transactions concurrently will **improve database performance**
  - **Increase throughput** (*average number of completed transactions*)
    - For example, while one transaction is waiting for an object to be read from disk, the CPU can process another transaction (because I/O activity can be done in parallel with CPU activity).
  - **Reduce latency** (*average time to complete a transaction*)
    - For example, interleave execution of a short transaction with a long transaction usually allows the short one to be completed more quickly.
- But the DBMS has to guarantee that the interleaving of transactions **does not lead to inconsistencies**, i.e., **concurrency control**.

Why is Concurrency Control Needed?

- Concurrency control is needed for preventing the following problems:

1. The **lost update** problem
2. The **dirty read** problem
3. The **unrepeated read** problem
4. The **phantom read** problem

(1) - The Lost Update Problem

- **Example:** Bob withdraws **$100** from his account ($T_1$) while Alice deposits **$500** into Bob's account (T_2).

```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T1: COMMIT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```

Steps	T_1	T_2
1	read(B)	
2		read(B)
3	write(B) (B:=B-100)	
4	commit	
5		write(B) (B:=B+500)
6		commit

Steps	B(Bob)
before 1	$200
after 2	$200
after 4	$100
after 6	$700

- **Question:** What is the problem?

(1) - The Lost Update Problem

- **Example:** Bob withdraws **$100** from his account ($T_1$) while Alice deposits **$500** into Bob's account (T_2).

```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T1: COMMIT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```

Steps	T_1	T_2
1	read(B)	
2		read(B)
3	write(B) (B:=B-100)	
4	commit	
5		write(B) (B:=B+500)
6		commit

Steps	B(Bob)
before 1	$200
after 2	$200
after 4	$100
after 6	$700

- **Answer:** Bob's balance should be **$600**. The update by T_1 is lost!

(1) - The Lost Update Problem

- Occurs when two transactions update the same object, and one transaction could overwrite the value of the object which has already been updated by another transaction (**write-write conflicts**).
- **Example:**

T_1	T_2
$read(B)$	
	$read(B)$
$write(B)$ $(B := B - 100)$	
$commit$	
	$write(B)$ $(B := B + 500)$
	$commit$

- $write(B)$ by T_2 overwrites B, and the update by T_1 is *lost*.

(2) - The Dirty Read Problem

- **Example:** Bob withdraws **$100** from his account ($T_1$) while Alice deposits **$500** into Bob's account (T_2).

```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: ABORT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```

Steps	T_1	T_2
1	read(B)	
2	write(B) (B:=B-100)	
3		read(B)
4	abort	
5		write(B) (B:=B+500)
6		commit

Steps	B(Bob)
before 1	$200
after 1	$200
after 2	$100
after 4	$200
after 6	$600

- **Question:** What is the problem?

(2) - The Dirty Read Problem

- **Example:** Bob withdraws **\$100** from his account (T_1) while Alice deposits **\$500** into Bob's account (T_2).

```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: UPDATE ACCOUNT SET balance=balance-100 WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T1: ABORT;
T2: UPDATE ACCOUNT SET balance=balance+500 WHERE name='Bob';
T2: COMMIT;
```

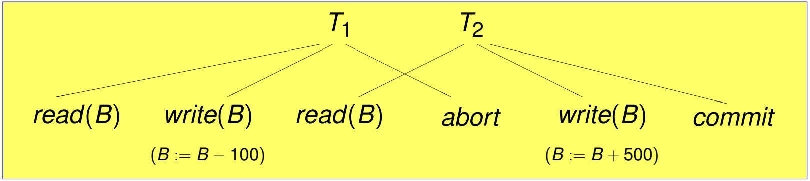
Steps	T_1	T_2
1	read(B)	
2	write(B) (B:=B-100)	
3		read(B)
4	abort	
5		write(B) (B:=B+500)
6		commit

Steps	B(Bob)
before 1	\$200
after 1	\$200
after 2	\$100
after 4	\$200
after 6	\$600

- **Answer:** Bob's balance should be **\$700** since T_1 was not completed.

(2) - The Dirty Read Problem

- Occurs when one transaction could read the value of an object that has been updated by another transaction but has not yet committed (**write-read conflicts**).
- **Example:**



- T_1 fails and must change the value of B back to **\$200**; but T_2 has read the uncommitted ($\cong$ *dirty*) value of B (**\$100**).

(3) - The Unrepeatable Read Problem

- **Example:** Bob checks his account (T_1) twice (takes time to decide whether to withdraw **$200**) while Alice withdraws **$500** from Bob's account (T_2).

```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: UPDATE ACCOUNT SET balance=balance-500 WHERE name='Bob';
T2: COMMIT;
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
```

Steps	T_1	T_2
1	read(B)	
2		read(B)
3		write(B) (B:=B-500)
4		commit
5	read(B)	

Steps	B(Bob)
before 1	$500
after 2	$500
after 3	$0
after 4	$0
after 5	$0

- **Question:** What is the problem?

(3) - The Unrepeatable Read Problem

- **Example:** Bob checks his account (T_1) twice (takes time to decide whether to withdraw **$200**) while Alice withdraws **$500** from Bob's account (T_2).

```
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: SELECT balance FROM ACCOUNT WHERE name='Bob';
T2: UPDATE ACCOUNT SET balance=balance-500 WHERE name='Bob';
T2: COMMIT;
T1: SELECT balance FROM ACCOUNT WHERE name='Bob';
```

Steps	T_1	T_2
1	read(B)	
2		read(B)
3		write(B) (B:=B-500)
4		commit
5	read(B)	

Steps	B(Bob)
before 1	$500
after 2	$500
after 3	$0
after 4	$0
after 5	$0

- **Answer:** Bob received two different account balances **$500** and **$0**, even though he hasn't withdrawn any money yet.

(3) - The Unrepeatable Read Problem

- A transaction could change the value of an object that has been read by another transaction but is still in progress (could issue two read for the object, or a write after reading the object) (**read-write conflicts**).
- **Example:**

T_1 T_2

$read(B)$ $read(B)$ $write(B)$ $commit$ $read(B)$ $\cdots$

$(B = 500)$ $(B = 500)$ $(B = 0)$

(4) - The Phantom Read Problem

- **Example:** A query is submitted for finding all customers whose account balances are less than **$300** ($T_1$) while Alice is opening a new account with the balance **$200** ($T_2$).
- Assume that only Bob (B) has an account whose balance is less than **$300** before Alice (A) opens his new account.

```
T1: SELECT name FROM ACCOUNT WHERE balance<300;
T2: INSERT INTO ACCOUNT(id, name, balance) VALUES(99, 'Alice', 250);
T2: COMMIT;
T1: SELECT name FROM ACCOUNT WHERE balance<300;
```

Steps	T_1	T_2
1	read(R)	
2		write(R)
3		commit
4	read(R)	

Steps	Query result
before 1	$R = \{B\}$
after 1	$R = \{B\}$
after 2	$R = \{A, B\}$
after 4	$R = \{A, B\}$

- **Question:** What is the problem?

(4) - The Phantom Read Problem

- **Example:** A query is submitted for finding all customers whose account balances are less than **$300** ($T_1$) while Alice is opening a new account with the balance **$200** ($T_2$).
- Assume that only Bob (B) has an account whose balance is less than **$300** before Alice (A) opens his new account.

```
T1: SELECT name FROM ACCOUNT WHERE balance<300;
T2: INSERT INTO ACCOUNT(id, name, balance) VALUES(99, 'Alice', 250);
T2: COMMIT;
T1: SELECT name FROM ACCOUNT WHERE balance<300;
```

Steps	T_1	T_2
1	read(R)	
2		write(R)
3		commit
4	read(R)	

Steps	Query result
before 1	$R = \{B\}$
after 1	$R = \{B\}$
after 2	$R = \{A, B\}$
after 4	$R = \{A, B\}$

- **Answer:** T_1 reads `Account` based on the condition `balance<300` twice but gets two different results $\{B\}$ and $\{A, B\}$.

(4) - The Phantom Read Problem

- Occurs when tuples updated by a transaction T_1 satisfy the search conditions of another transaction so that, by the same search condition, the transaction obtains different results at different times.
- **Example:**

	T_1	T_2	
$read(R)$	$write(R)$	$commit$	$read(R)$
$(R = \{B\})$	$(insert\ A)$		$(R = \{A, B\})$