Transactions and Access Control

Lecturer: Neena Thota
neena.thota@it.uu.se

Intended Learning Outcomes

- Write SQL programs (transactions).
- Understand the expected properties of SQL programs (ACID).
- Describe the problems prevented by ACID properties.
- Set different isolation levels.
- Control access to database objects.

Transactions: the setting

- Database systems are normally being accessed by many users or processes at the same time
  - Both queries and modifications
- To fulfill user requirements the system may need to execute a sequence of operations (queries and updates) and not just single operations.
- **Transaction** definition: An executing program that forms a logical unit of database processing.

Transactions: an example

```
SELECT Balance INTO balance_acct_1
FROM Accounts
WHERE AcctNo = account1;
```

1. read(A)

```
balance_acct_1 := balance_acct_1 – 50
```

2. A := A - 50

```
UPDATE Accounts
SET Balance = balance_acct_1
WHERE AcctNo = account1;
```

3. write(A)

```
SELECT Balance INTO balance_acct_2
FROM Accounts
WHERE AcctNo = account2;
```

4. read(B)

```
balance_acct_2 := balance_acct_2 + 50
```

5. B := B + 50

```
UPDATE Accounts
SET Balance = balance_acct_2
WHERE AcctNo = account2;
```

6. write(B)

Potential problem 1: Consistency

1. read(A)
2. A := A - 50
3. write(A)
4. read(B)
5. **B := B - 50**
6. write(B)

- We assume that a transaction starting from a consistent database leaves the database in a **consistent state**.
- What is a transaction has **interference** from other transactions before being completely executed from beginning to end.
- Need be aware of **who can write which SQL statements on which table.**

Potential problem 2: Atomicity

1. read(A)
2. A := A - 50
3. write(A)

- During the transaction execution some **inconsistent states** can be created.
- We can even allow schema constraints not to be satisfied during the transaction, as far as they are **satisfied at the end**.
- What if the transaction is **interrupted in the middle**?

UPPSALA UNIVERSITET

Potential problem 3: Durability

1. read(A)
2. A := A - 50
3. write(A)
4. read(B)
5. B := B + 50
6. write(B)

- After the transaction has been executed, its effect **must be permanent**.
- The changes applied to the database by a committed transaction must **persist in the database.**

Potential problem 4: Isolation

1. read(A)	
2. read(B)	1. read(B)
3. A := A - 50	
4. B := B - 50	2. B := B - 50
5. write(A)	
6. read(B)	
7. write(B)	3. write(B)
8. B := B + 50	
9. read(A)	4. read(A)
10. A := A + 50	5. A := A + 50
11. write(B)	
12. write(A)	6. write(A)

What is the result of this execution, if initially both A and B contain 100$?

ACID properties of transactions

- **Atomic**: either the whole transaction is executed, or none is.
- **Consistent**: database constraints are preserved.
- **Isolated**: appears to the user as if only one process executes at a time.
- **Durable**: effects of a transaction do not get lost if the system crashes.

Transactions in SQL

- SQL supports transactions, often behind the scenes
  - Each statement issued on a generic query interface is a transaction by itself
- **BEGIN** or **START TRANSACTION**
- **COMMIT** causes a transaction to complete.
  - Atomicity, Consistency, Isolation and Durability.
- **ROLLBACK** also causes the transaction to end, but by *aborting*.
  - No effects on the database.
  - Failures like division by 0 can also cause rollback, even if the programmer does not request it.
- **END TRANSACTION**

UPPSALA UNIVERSITET

ISOLATION

Our example as a transaction

```
BEGIN;

SELECT Balance INTO balance_acct_1
FROM Accounts
WHERE AcctNo = account1;

IF balance_acct_1<0 THEN ROLLBACK;
END IF;

balance_acct_1 := balance_acct_1 – 50;

UPDATE Accounts
SET Balance = balance_acct_1
WHERE AcctNo = account1;

-- etc. etc. etc.

COMMIT;
```

Isolation Levels

- SQL defines four isolation levels: choices about what interactions are allowed by transactions that execute at about the same time.

Within a transaction, we can say:

SET TRANSACTION ISOLATION LEVEL X

where X =

SERIALIZABLE

REPEATABLE READ

READ COMMITTED

READ UNCOMMITTED

Levels of isolation

1. **Serializable** — ACID E.g. airline reservation, debit credit, salary increase etc.
2. **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. *However, a transaction may find some records inserted by a transaction but not find others*. E.g. average value of one column, where we do not need a precise result but just an estimate.
3. **Read committed** — only committed records can be read, *but successive reads of a record may return different (but committed) values*. *E.g*. making balance, weather, departure or arrival times
4. **Read uncommitted** — even uncommitted records may be read. E.g. statistical average of a large number of data is required.

- Lower degrees of consistency useful for gathering approximate information about the database, e.g. statistics for query optimizer.
- Your choice, e.g., running a transaction under repeatable read isolation level, affects only how you see the database, not how others see it.

Isolation levels: summary

Get more tuples when we re-read

Get less tuples or different values when we re-read

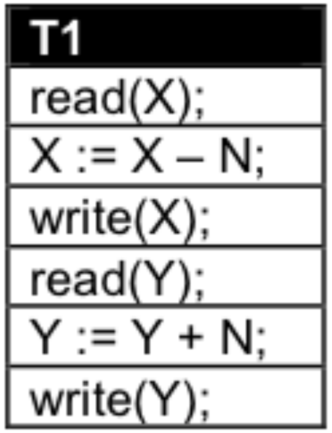
Get values that should not exist

Problems / Level	Dirty Read	Unrepeatable Read	Phantom
READ UNCOMMITTED	Possible	Possible	Possible
READ COMMITTED	Not possible	Possible	Possible
REPEATABLE READ	Not possible	Not possible	Possible
SERIALIZABLE	Not possible	Not possible	Not possible

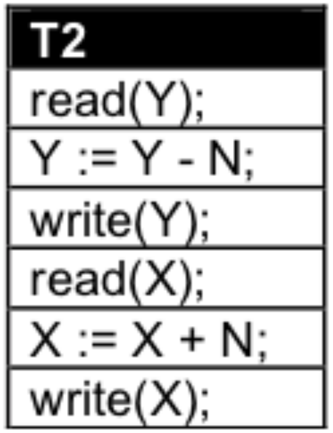
Example 1

Serial	Serializable	Non serializable
1. read(A)	1. read(A)	1. read(A)
2. A := A - 50	1. read(B)	2. A := A - 50
3. write(A)	2. B := B – 50	3. write(A)
4. read(B)	3. write(B)	1. read(B)
5. B := B + 50	2. A := A - 50	4. read(B)
6. write(B)	3. write(A)	5. B := B + 50
1. read(B)	4. read(B)	6. write(B)
2. B := B – 50	5. B := B + 50	2. B := B – 50
3. write(B)	6. write(B)	3. write(B)
4. read(A)	4. read(A)	4. read(A)
5. A := A + 50	5. A := A + 50	5. A := A + 50
6. write(A)	6. write(A)	6. write(A)
A=100, B=100	A=100, B=100	A=100, B=50

Example 2



T1
read(X);
X := X – N;
write(X);
read(Y);
Y := Y + N;
write(Y);



T2
read(Y);
Y := Y - N;
write(Y);
read(X);
X := X + N;
write(X);

Consider transactions
T1 and T2:
We represent as follows:
T1 : r1 (X); w1 (X); r1 (Y); w1 (Y);
T2 : r2 (Y); w2 (Y); r2 (X); w2 (X);

Question: Is the following schedule serializable? Why?

r1 (X); w1 (X); r1 (Y); r2 (Y); w1 (Y);
w2 (Y); r2 (X); w2 (X);

T1	T2
r1 (X);	
w1 (X);	
r1 (Y);	
	r2 (Y);
w1 (Y);	
	w2 (Y);
	r2 (X);
	w2 (X);

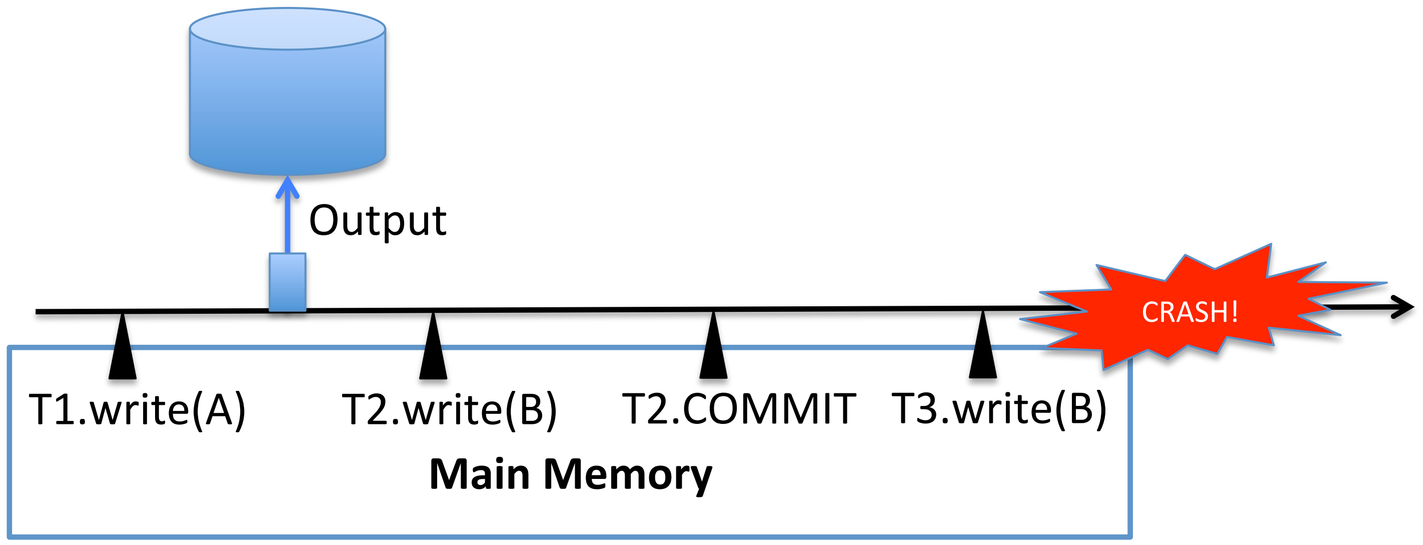
No, because T1 modifies Y after T2 has read it, so the final value of Y will depend on __?

UPPSALA UNIVERSITET

ATOMICITY AND DURABILITY

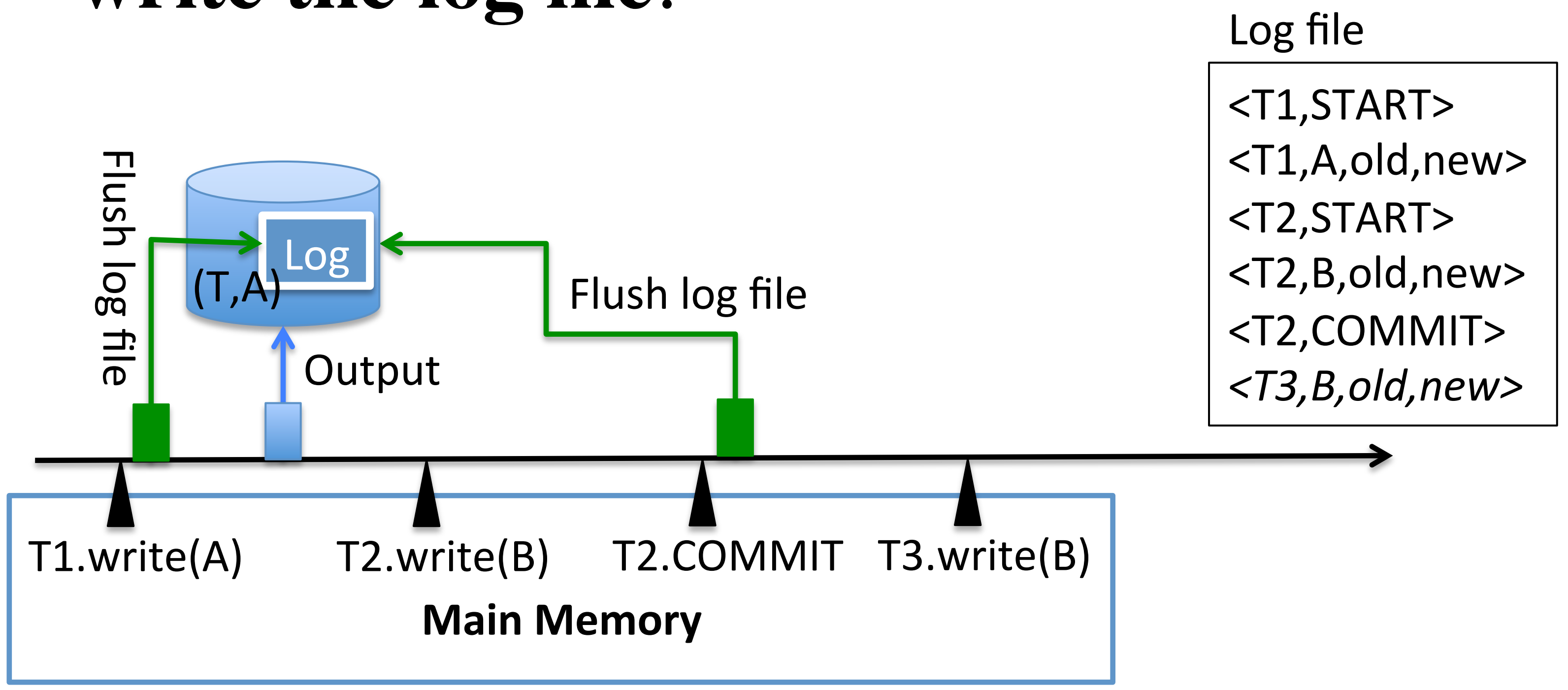
Problem setting

- When we execute a transaction, it modifies data in **main memory**.
- Data are saved to disk only after some operations have been performed, for efficiency reasons (**disks are slow**).



Recovery through logging

- We can store the description of the operations on a **log file**.
- Before performing operations on disk, we **write the log file**.



Database preservation

- Logging only prevents problems related to de-synchronization with main memory.
- It is normal procedure to store logs or full snapshots of the DB (= archiving, Back Up) remotely.

UPPSALA UNIVERSITET

Distributed Databases

- Process unit of execution (a transaction) in a **distributed manner**.
- A transaction can be executed by multiple networked computers in a **unified manner**.
- ***Problems***
  - Distributed commit
  - Distributed deadlock
- ***Solutions***
  - Primary site technique: A single site is designated as a primary site which serves as a coordinator for transaction management.
  - Concurrency control and commit are managed by this site.

UPPSALA UNIVERSITET

CONSISTENCY

UPPSALA UNIVERSITET

GRANT

GRANT *privileges* | ALL PRIVILEGES

ON *object*

TO *list of users | list of roles |* PUBLIC

[WITH GRANT OPTION]

UPPSALA UNIVERSITET

GRANT: comments

- Available privileges:
  - SELECT, UPDATE and INSERT can target specific columns.
  - Other privileges for other schema objects exist – not mentioned here.
- Privileges are given to users and to ***roles***.

GRANT *list of roles*
TO *list of users* | *list of roles* | PUBLIC
[**WITH ADMIN OPTION**]

GRANT: examples

- GRANT update(telephone) ON Clients TO Elisa;
- GRANT usage ON TYPE Address TO John **WITH GRANT OPTION**;
- GRANT select(name, surname)

  ON Clients TO Elisa;
- GRANT delete, update ON Clients TO admin_emp;
- GRANT admin_emp TO Robert **WITH ADMIN OPTION**;

REVOKE

```
REVOKE
[GRANT OPTION FOR | HIERARCHY OPTION FOR]
privileges
ON object
FROM list of users | list of roles
RESTRICT | CASCADE

REVOKE [ADMIN OPTION FOR] list of roles
FROM list of users | list of roles
RESTRICT | CASCADE
```

UPPSALA UNIVERSITET

REVOKE: comments

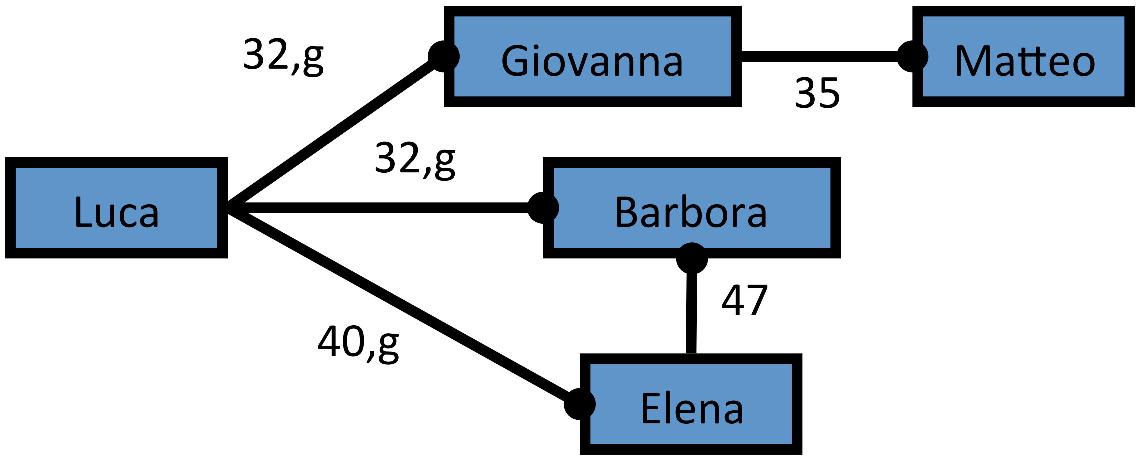
- We can revoke both privileges and roles.
- We can revoke GRANT | ADMIN OPTIONS without revoking the corresponding privileges.
- If **RESTRICT** is specified, the command is not executed if it would determine recursive revoke actions.
- With **CASCADE**, revoke actions are propagated so that the *graph of grants* remain consistent.

REVOKE: examples

- REVOKE delete **ON** Clients **FROM** admin_emp;
- REVOKE admin_emp **FROM** Robert;

Recursive revoke

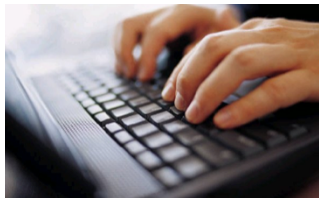
- Luca revokes from Giovanna.
  - What happens to Matteo? (he loses the privilege)
- Luca revokes from Elena
  - What happens to Barbora?



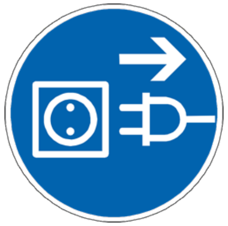
Views to control access

- Views and GRANT/REVOKE are the main way to **prevent unwanted modifications or access** to relational databases.
- **Views** define parts of the data that can be made visible to specific users or roles, and privileges are assigned on the views.

A summary of problems and solutions

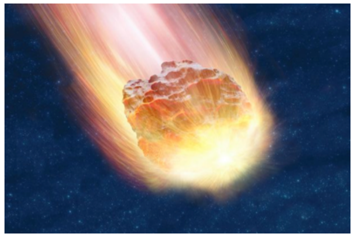

Wrong data entry

Malicious data entry


Loss of main memory

Broken disks

Data center destruction


Large scale natural catastrophe

A) Logging
B) Normalization
C) Remote archiving
D) Check constraints
E) Back up
F) Authorizations