Database Usage (and Construction)

Transactions

Authorization

Setting

- DBMS must allow concurrent access to databases.
  - Imagine a bank where account information is stored in a database *not* allowing concurrent access. Then only one person could do a withdrawal in an ATM machine at the time – anywhere!
- Uncontrolled concurrent access may lead to problems.

Example:

Imagine a program that does the following:

1. Get a day, a time and a course from the user in order to schedule a lecture. **`(get)`**

2. List all available rooms at that time, with number of seats, and let the user choose one. **`(list)`**

3. Book the chosen room for the given course at the given time. **`(book)`**

```
SELECT *
FROM ROOMS
WHERE name NOT IN
  (SELECT room
   FROM Lectures
   WHERE weekday = theDay
     AND hour = theTime);
```

```
INSERT INTO Lectures VALUES
  (theCourse, thePeriod,
   theDay, theTime,
   chosenRoom);
```

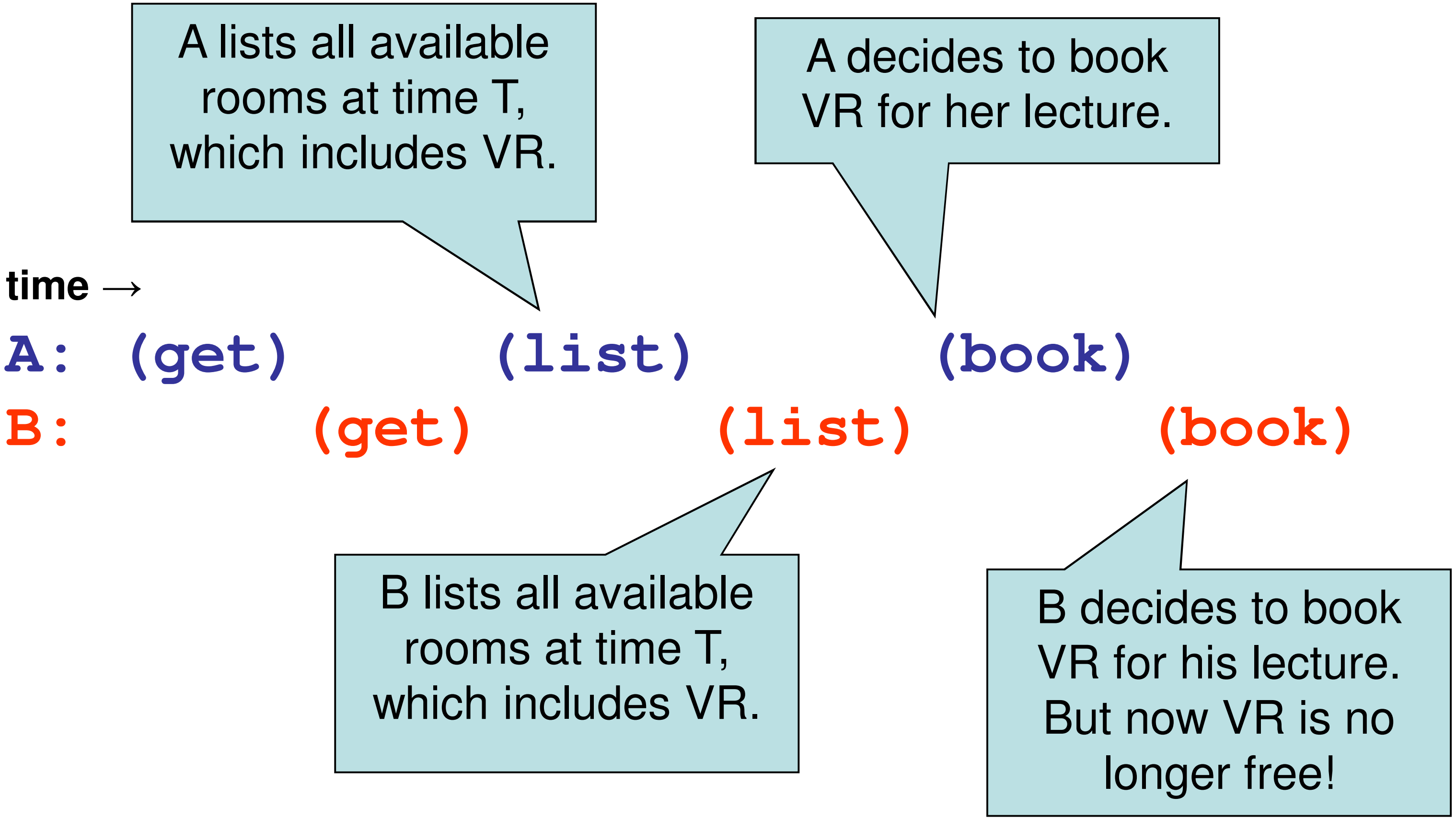
What could go wrong?

Running in parallel

- Assume two people, A and B, both try to book a room for the same time, at the same time.
- Both programs perform the sequence **`(get) (list) (book)`**, in that order.
- But we can interleave the blocks of the two sequences in any way we like!
  - Here’s one possible interleaving:

```
A:  (get)       (list)        (book)
B:        (get)        (list)        (book)
```

Interleaving



A lists all available rooms at time T, which includes VR.

A decides to book VR for her lecture.

time →

```
A:  (get)       (list)       (book)
B:       (get)       (list)       (book)
```

B lists all available rooms at time T, which includes VR.

B decides to book VR for his lecture. But now VR is no longer free!

DBMS vs OS

- An operating system supports concurrent access, and interaction.
  - E.g. two users modify the same file. If both save their changes, then the changes of one get lost.
- A DBMS must support concurrent access, but must keep processes from interacting!

Quiz!

Look again at the interleaving:

time →

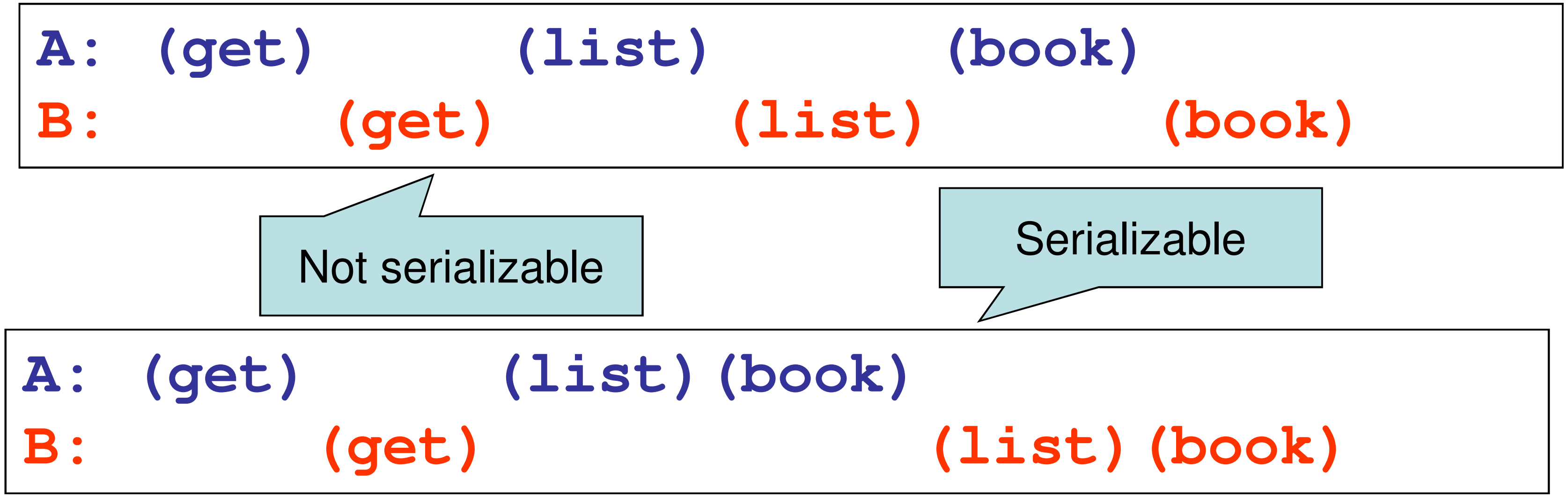
```
A:  (get)      (list)        (book)
B:        (get)       (list)        (book)
```

What can we do to fix it?

The only way that we get the desired behavior is if both A and B may perform the operations **`(list) (book)`** without the other doing a **`(book)`** in the middle!

Serializability

- Two programs are run *in serial* if one finishes before the other starts.
- The running of two programs is *serializable* if the effects are the same as if they had been run in serial.



```
A:  (get)       (list)          (book)
B:        (get)        (list)          (book)
```

Not serializable

Serializable

```
A:  (get)       (list) (book)
B:        (get)               (list) (book)
```

Example:

Assume we perform the following operations to transfer 100 SEK from account X to account Y.

1. Check account balance in account X.

```
SELECT balance
FROM Accounts
WHERE accountID = X;
```

2. Subtract 100 from account X.

```
UPDATE Accounts
SET balance = balance – 100
WHERE accountID = X;
```

3. Add 100 to account Y.

```
UPDATE Accounts
SET balance = balance + 100
WHERE accountID = Y;
```

Two things can go wrong: We can have strange interleavings like before. But also, assume the program crashes after executing 1 and 2 – we'll have lost 100 SEK!

Quiz!

Assume we run the following two programs in parallel, and assume the databases contains only the Databases lecture in room VR on Mondays (and all lectures are 2h long):

P_1:

1. Insert a lecture for the Databases course in room VR at 10 on Mondays. **`(ins)`**
2. Delete the lecture in the Databases course in room VR at 13 on Mondays. **`(del)`**

P_2:

1. Find the first lecture of the day in room VR on Mondays. **`(min)`**
2. Find the last lecture of the day in room VR on Mondays. **`(max)`**
3. Return the total time that room VR is occupied, *((max+2)-min).* **`(ret)`**

...Quiz continued!

P_1	`(ins) (del)`	What could P_2 return?
P_2	`(min) (max) (ret)`	

- Need to consider possible **schedules** of the actions that access or update the database: `(ins) (del) (min) (max)`

`(ins) (del) (min) (max)`	P_2 returns 2
`(ins) (min) (del) (max)`	P_2 returns 2
`(ins) (min) (max) (del)`	P_2 returns 5
`(min) (ins) (del) (max)`	P_2 returns -1
`(min) (ins) (max) (del)`	P_2 returns 2
`(min) (max) (ins) (del)`	P_2 returns 2

ACID Transactions

- A DBMS is expected to support ”ACID transactions”, which are
  - **Atomic**: Either the whole transaction is run, or nothing.
  - **Consistent**: Database constraints are preserved.
  - **Isolated**: Different transactions may not interact with each other.
  - **Durable**: Effects of a transaction are not lost in case of a system crash.

Atomicity

- For many programs, we require that ”all or nothing” is executed.
  - We say a sequence of actions is executed *atomically* if it is executed either in entirety, or not at all.
    - The state in the middle is never visible from outside the sequence.
    - cf. Greek atom = indivisible.
    - In case of a crash in the middle, any changes that were made up until that point must be undone.

Transactions in SQL

- SQL supports transactions, often behind the scenes.
  - An SQL statement is a transaction.
    - E.g. an update of a table can’t be interrupted after half the rows.
    - Any triggers, procedures, functions etc. that are started by the statement is part of the same transaction.
  - In PSM or Embedded SQL, a transaction begins at the first SQL operation and ends when the program does, or it is explicitly ended by the programmer.

Controlling transactions

- We can explicitly start transactions using the **`START TRANSACTION`** statement, and end them using **`COMMIT`** or **`ROLLBACK`**:
  - **`COMMIT`** causes an SQL transaction to complete successfully.
    - Any modifications done by the transaction are now permanent in the database.
  - **`ROLLBACK`** causes an SQL transaction to end by aborting it.
    - Any modifications to the database must be undone.
    - Rollbacks could be caused implicitly by errors e.g. division by 0.

Read-only vs. Read-write

- A transaction that does not modify the database is called *read-only*.
  - A read-only transaction can never interfere with another transaction (but not the other way around!).
  - Any number of read-only transactions can be run concurrently.
- A transaction that both reads and modifies the database is called *read-write*.
  - No other transaction may write between the read and write.

SET TRANSACTION

- We can hint the DBMS that a transaction only does reading, by issuing the statement:

```
SET TRANSACTION READ ONLY;
```

  - Possibly the DBMS can make use of the information and optimize scheduling.

Drawbacks

- Serializability and atomicity are necessary, but don’t come without a cost.
  - We must retain old data until the transaction commits.
  - Other transactions may need to wait for one to complete.
- In some cases some interference may be acceptable, and could speed up the system greatly.

Example:

Recall the first example of booking rooms:

```
time →
A:  (get)      (list)       (book)
B:       (get)        (list)       (book)
```

It could take time for the user to decide which room to choose after getting the list. If we make this a serializable transaction, all other users would have to wait as well.

The worst thing that could happen is that B is told to choose another room when he tries to book the room that A just booked.

Isolation levels

- ANSI SQL standard defines four *isolation levels*, which are choices about what kinds of interference are allowed between transactions.
- Each transaction chooses its own isolation level, deciding how other transactions may interfere with it.
- Isolation level is defined in terms of three phenomena that can occur.

Kinds of interference

The ANSI SQL standard describes:

- Dirty read
- Non-repeatable read
- Phantom

(These, and other kinds of interference, are discussed in: Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., & O'Neil, P. (1995). A critique of ANSI SQL isolation levels. ACM SIGMOD Record, 24(2), 1-10.)

Dirty read

- Transaction T1 modifies a data item.
- Another transaction T2 then reads that data item before T1 performs a COMMIT or ROLLBACK.
- If T1 then performs a ROLLBACK, T2 has read a data item that was never committed and so never really existed.

Non-repeatable read

- Transaction T1 reads a data item.
- Another transaction T2 then modifies or deletes that data item and commits.
- If T 1 then attempts to re-read the data item, it receives a modified value or discovers that the data item has been deleted.

Phantom

- Transaction T1 reads a set of data items satisfying some <search condition>.
- Transaction T2 then creates data items that satisfy T1's <search condition> and commits.
- If T1 then repeats its read with the same <search condition>, it gets a set of data items different from the first read.

Choosing isolation level

- Within a transaction we can choose the isolation level:

```
SET TRANSACTION ISOLATION LEVEL X;
```

where X is one of

- **SERIALIZABLE**
- **READ COMMITTED**
- **READ UNCOMMITTED**
- **REPEATABLE READ**

Isolation levels - differences

What kinds of interference are possible?

	Dirty reads	Non-repeatable reads	Phantoms
`READ UNCOMMITTED`	Yes	Yes	Yes
`READ COMMITTED`	No	Yes	Yes
`REPEATABLE READ`	No	No	Yes
`SERIALIZABLE`	No	No	No

SERIALIZABLE

- If a transaction is run with isolation level **SERIALIZABLE**, then no other transaction may interfere with it in any way.
  - Examples:

If two room booking transactions are run serializable, then a booking for a room that was listed as free will always succeed, and transactions must wait for other transactions to finish.

In the min-max example, we always get a value that is correct at some point in time, either before or after the updating.

READ COMMITTED

- If a transaction is run with isolation level **READ COMMITTED**, then the transaction allows other transactions to modify the database while running.
- Anything that is committed by another transaction affects the reads of this transaction.

Quiz!

If we run two room booking transactions, **(list) (book)**, in parallel with isolation level **READ COMMITTED**, what would happen?

One transaction could book a room after the other had listed it as free, and the second booking may fail.
On the other hand, no transaction must wait for any other to finish.

Quiz again!

If we run the first transactions of the min-max example, (`(min)` `(max)` and `(ins)` `(del)`), as **`READ COMMITTED`**, what could happen?

The update could be done between min and max, which means we could get the value -1.
If the updating is run **`SERIALIZABLE`**, we could not see the state between since the changes would not be committed, so the value 5 is not possible.

READ UNCOMMITTED

- If a transaction is run with isolation level **READ UNCOMMITTED**, then the transaction allows other transactions to modify the database while running.
- Anything that is changed by another transaction affects the reads of this transaction, even if the other transaction has not yet committed!

Quiz!

If we extend the room booking transaction with a confirmation, i.e. **`(list)(book)(confirm)`**, and run two in parallel with isolation level **`READ UNCOMMITTED`**, what could happen?

Same as with **`READ COMMITTED`**, except that if the user of the first transaction changes her mind at confirmation, thus causing a roll-back, the second user could be told that the room is booked even though it never was!

Quiz again!

If we run the first transactions of the min-max example as **`READ UNCOMMITTED`**, what could happen?

The update could be done between **`(min)`** and **`(max)`**, which means we could get the value -1.
Even if the updating is run **`SERIALIZABLE`**, we could see the state between **`(ins)`** and **`(del)`**, so the value 5 is also possible in this case!

Remember: Isolation level is a personal choice. Only because the min-max transaction is read-only can we run it in the middle of a serializable transaction!

REPEATABLE READ

- If a transaction is run with isolation level **REPEATABLE READ**, it works like read committed, except:
- If the transaction reads more than once, we are guaranteed to get *at least* the same tuples again (though we could get more).

Quiz!

If we run two room booking transactions, **`(list)`** **`(book)`**, in parallel with isolation level **`REPEATABLE READ`**, what would happen?

Exactly the same thing as for **`READ COMMITTED`**, since we only read once!

Quiz again!

If we run the first transactions of the min-max example as **`REPEATABLE READ`**, what could happen?

If the update is done between **`(min)`** and **`(max)`**, we will still see the deleted value when doing **`(max)`**, so we can only get the value 2.

… but if we do (max)(min) instead, we would get the value 5…

Summary transactions

- DBMS must ensure that different processes don’t interfere with each other!
  - ”ACID”: Atomicity, Consistency, Isolation, Durability.
  - The isolation levels of transactions may vary.
    - Serializable
    - Read Committed
    - Read Uncommitted
    - Repeatable Read
  - Isolation level affects only that transaction!

Exam – Transactions

”Here are some transactions that run in parallel. …”

- What will the end results given by the transactions be?
- What could happen if they were not run as transactions?

Database Authorization

Authorization

- Not every user can be allowed to do everything.
  - Some data are secret and may only be seen by some users.
  - Some data are high integrity and may only be modified by certain users.

Database vs file system

- A (UNIX) file system has:
  - Privileges on files.
  - Three different privileges: read, write, execute
  - Three levels of access: owner, group, all
- A database has:
  - Privileges on schema elements (tables, views, triggers, etc.)
  - Nine different privileges.
  - Any number of levels of access – each user can be given different access.

Quiz!

Name the nine different privileges!

```
SELECT
INSERT
DELETE
UPDATE
REFERENCE
TRIGGER
EXECUTE
USAGE
UNDER
```

Privileges

- **SELECT (*attributes*) ON *table***
  - Allows the user to select data from the specified table.
  - Can be parametrized on attributes, meaning the user may only see certain attributes of the table.
- **INSERT (*attributes*) ON *table***
  - Allows the user to insert tuples into the table.
  - Can be parametrized on attributes, meaning the user may only supply values for certain attributes of the table. Other attributes are then set to NULL.

Privileges

- **`DELETE ON`** ***`table`***
  - – Allows the user to delete tuples from the table.
  - – Cannot be parametrized on attributes.
- **`UPDATE (`*****`attributes`*****`) ON`** ***`table`***
  - – Allows the user to update data in the table.
  - – Parametrizing means the user may only update values of certain attributes.

Quiz!

What does the **`REFERENCE`** privilege on (attributes in) a table do, and why is it needed?

It allows a user to reference that table from foreign key constraints, checks and assertions.

It is needed since creating a foreign key constraint restricts update and deletion on the referenced table.

Also knowing some value exists in a table is not the same as knowing what values exist in that table…

Privileges

- **`TRIGGER ON`** ***`table`***
  - Allows the user to create triggers for events on that table.
- **`EXECUTE ON`** ***`procedure`***
  - Allows the user to execute the procedure or function, and use it in declarations.
- **`USAGE ON`** ***`type`***
  - Used for non-relation elements, e.g. types – allows a user to use these elements in declarations.
- **`UNDER ON`** ***`type`***
  - Used on types – allows a user to create a subtype of the given type.

Quiz!

What privileges are needed to perform the following insertion?

```
INSERT INTO Lectures(course, period, weekday)
 SELECT course, period, ’Monday’
 FROM   GivenCourses G
 WHERE  NOT EXISTS
          (SELECT course, period
           FROM   Lectures L
           WHERE  L.course = G.course
                  AND L.period = G.period
                  AND weekday = ’Monday’);
```

We need privileges **`INSERT`** on **`Lectures(course, period, weekday)`**, **`SELECT`** on **`GivenCourses(course, period)`**, and **`SELECT`** on **`Lectures(course, period, weekday)`**.

Quiz!

Assume we have written this trigger. What privileges are now needed in order to insert values into DBLectures?

```
CREATE TRIGGER AddDBLecture
INSTEAD OF INSERT ON DBLectures
REFERENCING NEW ROW AS new
FOR EACH ROW
INSERT INTO Lectures
 VALUES (’TDA357’, 2, new.weekday,
            new.hour, new.room);
```

INSERT ON DBLectures and nothing else. However, the user that created the trigger must also have **INSERT ON Lectures** and **TRIGGER ON DBLectures**.

EXECUTE and TRIGGER

- When writing a trigger, the body may perform selections and modifications.
  - The user who writes the trigger must have all the necessary privileges to perform those operations, plus the **`TRIGGER`** privilege.
  - The user that sets off the trigger needs only the privilege to perform the triggering event (e.g. an insertion). Everything that happens in the trigger is considered done by its creator.
- The same thing goes for procedures and functions – it is the privileges of the creator that decides what operations may be performed, and the user needs only **`EXECUTE`**.

Granting privileges

- You have all possible privileges on elements that you have created.
- You may grant privileges to other users on those elements.
  - A user is referred to by an *authorization ID*, which is typically a user name.
  - There is a special authorization ID, *public*
  - Granting a privilege to *public* makes it available to all users.

GRANT statement

- Granting a privilege in SQL:

```
GRANT list of privileges
ON element
TO list of authorization Ids;
```

  – Example:

```
GRANT SELECT(course, period, teacher)
ON    GivenCourses
TO    public;
```

WITH GRANT OPTION

- A user that can grant privileges on some element can choose to grant **`WITH GRANT OPTION`**.
  - The grantee can then grant this privilege further.
  - Example:

```
GRANT SELECT(course, period, teacher)
ON    GivenCourses
TO    nibro WITH GRANT OPTION;
```

Revoking privileges

- Privileges can be revoked with the inverse statement:

```
REVOKE list of privileges
ON     element
FROM   list of authorization Ids;
```

- Your grant of these privileges can no longer be used by these users to justify their use of the privilege.
  - But they may still have the privilege because they have it from another independent source.

Quiz!

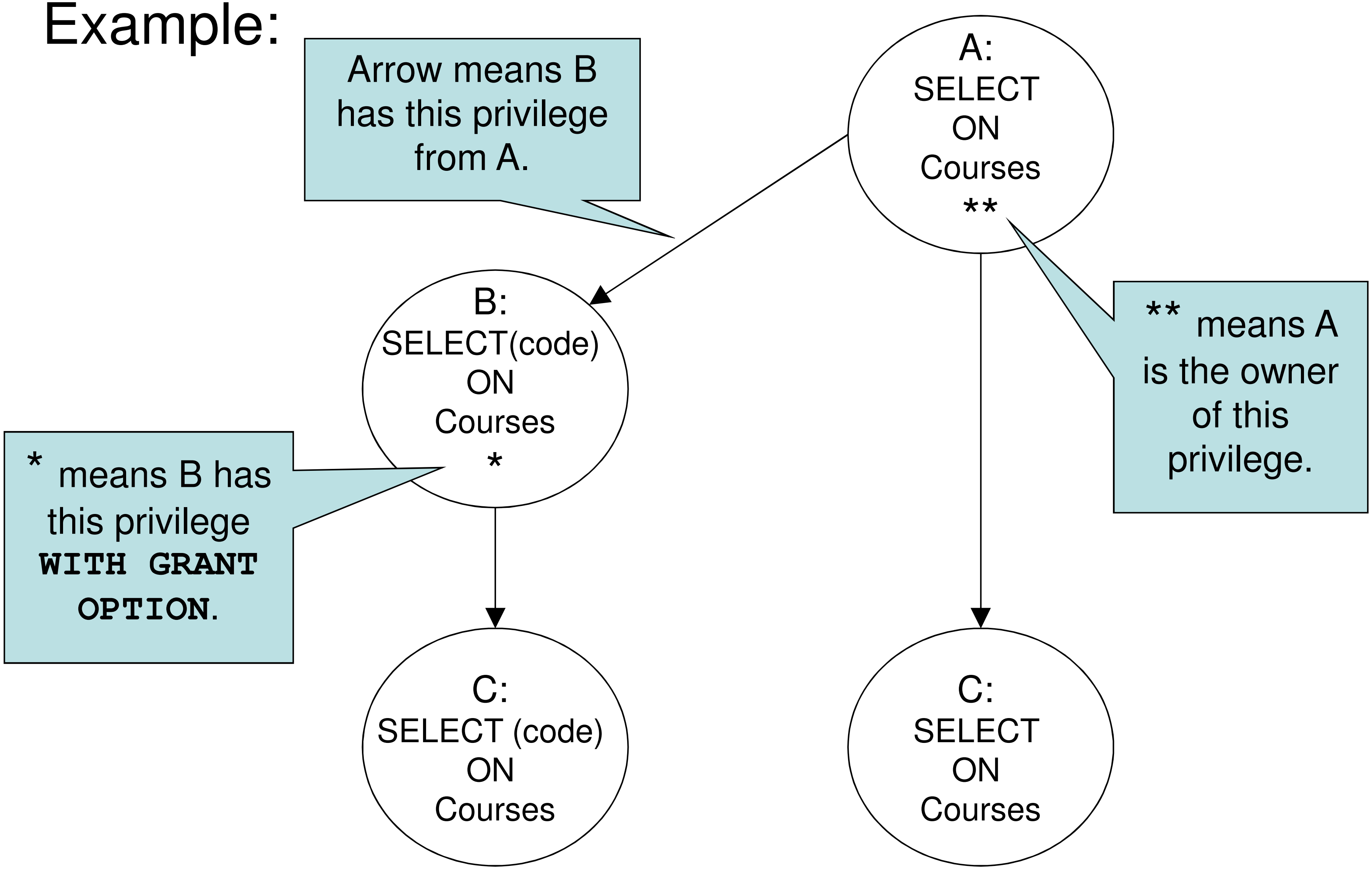
What happens if we revoke a privilege from a user who has it **`WITH GRANT OPTION`**, and who has given it further?

We have two choices: **`CASCADE`** or **`RESTRICT`**.
The first means we revoke the privilege from all those other users as well, while the latter means the revocation will fail with an error.
Cf. deleting rows from a table that is referenced.

Grant diagrams

- Nodes = user + privilege + option
  - Option is either owner, **`WITH GRANT OPTION`**, or neither.
  - **`UPDATE ON T`**, **`UPDATE(a) ON T`**, **`UPDATE(b) ON T`** and **`UPDATE ON T WITH GRANT OPTION`** all live in different nodes.
- Edge X → Y means that node X was used to grant Y.

Example:



A: SELECT ON Courses **

B: SELECT(code) ON Courses *

C: SELECT (code) ON Courses

C: SELECT ON Courses

Arrow means B has this privilege from A.

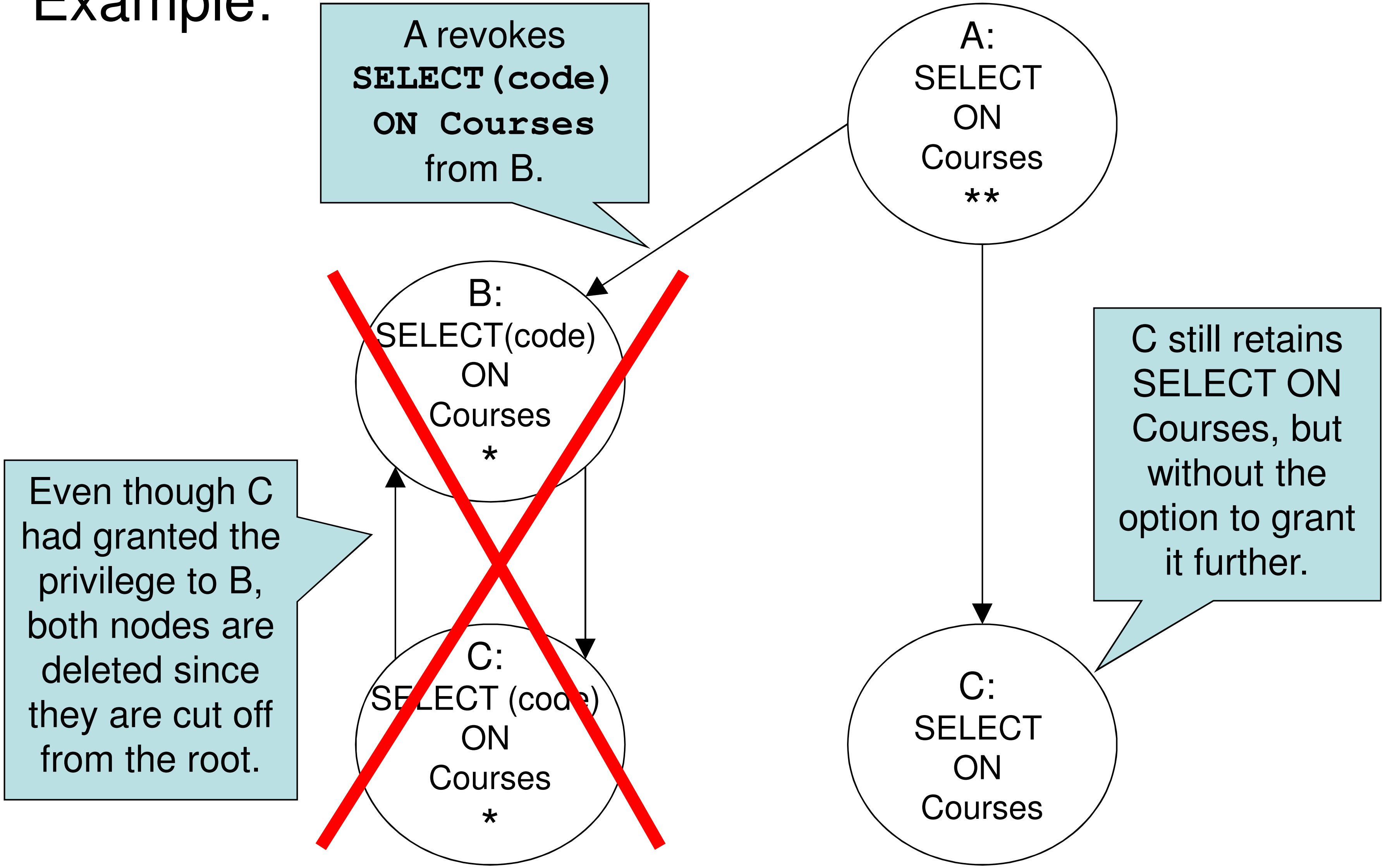
** means A is the owner of this privilege.

* means B has this privilege **WITH GRANT OPTION**.

Manipulating edges

- If A grants P to B, we draw an edge from AP* (or AP**) to BP(* if with grant option).
- Revoking a privilege means deleting the edge corresponding to the privilege.
- Fundamental rule: User U has privilege P as long as there is a path from XP** to either UP, UP* or UP**, where X is the owner of P.
  - Note that X could be U, in which case the path is 0 steps.

Example:



A revokes **`SELECT(code) ON Courses`** from B.

Even though C had granted the privilege to B, both nodes are deleted since they are cut off from the root.

C still retains SELECT ON Courses, but without the option to grant it further.

Summary Authorization

- Privileges in SQL
  - **`SELECT, INSERT, DELETE, UPDATE, REFERENCE, TRIGGER, EXECUTE`** …
- Granting and revoking privileges
  - Authentication IDs, public
  - **`WITH GRANT OPTION`**
- Grant diagrams