Ders # 6

Veri Bütünlüğü Programlama ve Güvenlik

From Standford's The complete Book

Sciore's textbook, Ch 9-10

From Raghu&Johennes DB textbook

Outline:

- General Constraints as Assertions
- Triggers
- Security

Objective:

❑ Specification of more general **constraints** via assertions, triggers
❑ Coordination of users access to data. Authorization issues on dabases.

Constraints as Assertions

- General constraints: constraints that do not fit in the basic SQL categories (*such as unique, primary key, not null, foreign key..*)

- Mechanism: **CREAT ASSERTION**
  - a constraint name,
  - followed by CHECK,
  - followed by a condition

- "The salary of an employee must not be greater than the salary of the manager of the department that the employee works for''

```
CREAT ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS ( SELECT *
                    FROM EMPLOYEE E, EMPLOYEE M, DEPARTMENT D
                    WHERE E.SALARY > M.SALARY AND
                    E.DNO=D.NUMBER AND
                    D.MGRSSN=M.SSN))
```

Examples on student record database:

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```

```
create assertion SmallSections
    check (not exists
            select e.SectionId
            from ENROLL e
            group by e.SectionId
            having count(e.EId) > 30)
```

Figure 5-1
The SQL specification of the integrity constraint
"No section can have more than 30 students"

Examples on student record database:

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```

```
create assertion ValidGradYear
    check (not exists
            select s.*
            from STUDENT s
            where s.GradYear < 1863)
```

Figure 5-2
The SQL specification of the integrity constraint
"A student's graduation year must be at least 1863"

Examples on student record database:

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```

```
create assertion NoTakeTwice
    check (not exists
            select e.StudentId, k.CourseId
            from SECTION k, ENROLL e
            where k.SectId=e.SectionId
            group by e.StudentId, k.CourseId
            having count(k.SectId)>1)
```

Figure 5-3
The SQL specification of the integrity constraint
"Students can take a course at most once"

SQL Triggers

- Objective: to monitor a database and take initiate action when a condition occurs

  - Triggers are action that fire automatically based on these conditions

- Triggers are expressed in a syntax similar to assertions and include the following:

  - Event : Such as an insert, deleted, or update operation
  - Condition : optional
  - Action:To be taken when the condition is satisfied

- CREATE TRIGGER <name>: Creates a trigger

- ALTER TRIGGER <name> : Alters a trigger (assuming one exists)

- CREATE OR ALTER TRIGGER <name>

  - Creates a trigger if one does not exist
  - Alters a trigger if one does exist

implementing a trigger

Can be BEFORE, AFTER, INSTEAD OF

CREATE TRIGGER *TrigName*

AFTER INSERT ON *TblName*

Can be INSERT, UPDATE, DELETE

FOR EACH ROW

WHEN (*condition*)

The condition

....

....

The action (like UPDATE, DELETE,..)

Valid Trigger Types

Triggered by INSERT:
- BEFORE INSERT statement-level.
- BEFORE INSERT row-level.
- AFTER INSERT statement-level.
- AFTER INSERT row-level.

Triggered by UPDATE:
- BEFORE UPDATE statement-level.
- BEFORE UPDATE row-level.
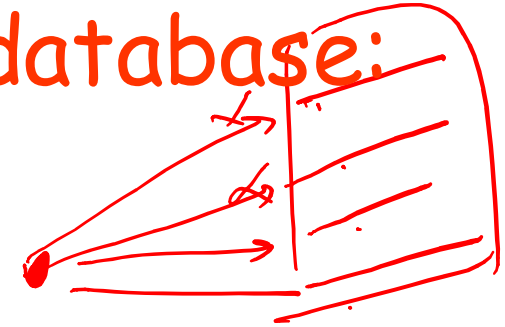- AFTER UPDATE statement-level.
- AFTER UPDATE row-level.

Triggered by DELETE:
- BEFORE DELETE statement-level.
- BEFORE DELETE row-level.
- AFTER DELETE statement-level.
- AFTER DELETE row-level.

To replace the triggering event:
- INSTEAD OF statement-level.
- INSTEAD OF row-level.

Examples on student record database:

```
create trigger LogGradeChange
    after update of Grade in ENROLL
    for each row
    referencing old row as oldrow, new row as newrow
    when oldrow.Grade <> newrow.Grade
        insert into GRADE_LOG(UserName, DateChanged, EId,
                                        OldGrade, NewGrade)
        values(current_user, current_date, newrow.EId,
                oldrow.Grade, newrow.Grade)
```

Figure 5-4
An SQL trigger that logs changes to student grades
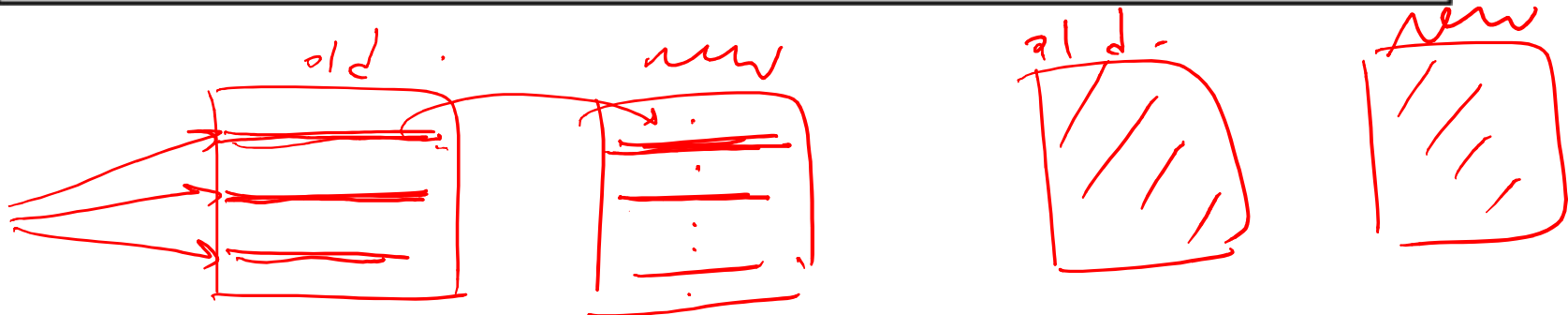
Examples on student record database:

```
create trigger FixBadGradYear
    after insert in STUDENT
    for each row
    referencing new row as newrow
    when newrow.GradYear > 4 + extract(YEAR, current_date)
        set newrow.GradYear = null
```

Figure 5-5
An SQL trigger that replaces inappropriate graduation years with nulls

fig_05_05

Triggers on Simplified COMPANY DB

- EMPLOYEE (NAME, SSN, SALARY, DNO, SUPERVİSOR_SSN)
- DEPARTMENT (DNAME, DNO, TOTAL_SAL, MANAGER _SSN)

- **EXAMPLE 1:** A trigger to compare an employee's salary to his/her supervisor during insert or update operations:

CREATE TRIGGER INFORM_SUPERVISOR
BEFORE INSERT OR UPDATE OF
    SALARY, SUPERVISOR_SSN ON EMPLOYEE
    FOR EACH ROW
      WHEN
      NEW.SALARY> (SELECT SALARY FROM EMPLOYEE
         WHERE SSN=NEW.SUPERVISOR_SSN))
      INFORM_SUPERVISOR (NEW.SUPERVISOR_SSN,NEW.SSN);

Triggers on Simplified COMPANY DB

EMPLOYEE (NAME, SSN, SALARY, DNO, SUPERVİSOR_SSN)

DEPARTMENT (DNAME, DNO, TOTAL_SAL, MANAGER _SSN)

- keep updated the DEPARTMENT.TOTAL_SAL

4 events runs the trigger:

- T1: Total_sal1 *(When a new employees is added to a department, modify the Total_sal of the Department to include the new employees salary)*
    - This trigger will execute **AFTER INSERT ON** Employee table
    - It will do the following FOR EACH ROW
        - WHEN NEW.Dno is NOT NULL
        - The trigger will UPDATE DEPARTMENT
        - By SETting the new Total_sal to be the sum of
            » old Total_sal and NEW. Salary
            » WHERE the Dno matches the NEW.Dno;

- T2:Total_sal2: (*changing salary of existing Emp.)*
- T3: (*Deleting an employee.)*
- T4: *(Changing department of an employee.)*

Implementing T1: «adding a new Employee»

T1: CREATE TRIGGER Total_sal1

AFTER INSERT ON Employee

FOR EACH ROW

WHEN (NEW.Dno is NOT NULL)

    UPDATE DEPARTMENT

    SET Total_sal = Total_sal + NEW. Salary

    WHERE Dno = NEW.Dno;

EVENT

CONDITION

The action

Implementing T2: «changing salary of existing Emp»

```
T2:  CREATE TRIGGER Total_sal2
     AFTER UPDATE OF Salary ON EMPLOYEE
     REFERENCING OLD ROW AS O, NEW ROW AS N
     FOR EACH ROW
     WHEN    ( N.Dno IS NOT NULL )
     UPDATE DEPARTMENT
     SET Total_sal = Total_sal + N.salary – O.salary
     WHERE Dno = N.Dno;
```

```
T2:  CREATE TRIGGER Total_sal2
     AFTER UPDATE OF Salary ON EMPLOYEE
     REFERENCING OLD TABLE AS O, NEW TABLE AS N
     FOR EACH STATEMENT
     WHEN    EXISTS  ( SELECT * FROM N WHERE N.Dno IS NOT NULL ) OR
             EXISTS  ( SELECT * FROM O WHERE O.Dno IS NOT NULL )
     UPDATE DEPARTMENT AS D
     SET D.Total_sal = D.Total_sal
         + ( SELECT SUM (N.Salary) FROM N WHERE D.Dno=N.Dno )
         – ( SELECT SUM (O.Salary) FROM O WHERE D.Dno=O.Dno )
     WHERE Dno IN ( ( SELECT Dno FROM N ) UNION ( SELECT Dno FROM O ) );
```

Cont... *T2 (for each statement)*

	Salary	DNO
	200	1
	300	2

EVENT: ONLY «SAL» changes !

Old O

	Salary	DNO
	200	1
	300	2

New N

	Salary	DNO
	700	1
	900	2
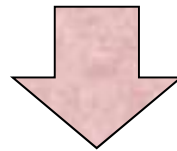
ACTION !

Cont... *T2 (for each statement)*

	Salary	DNO
	200	1
	300	2

EVENT: Both «SAL» and «DNO» changes !

Old O

	Salary	DNO
	200	1
	300	2

New N

	Salary	DNO
	700	2
	900	1
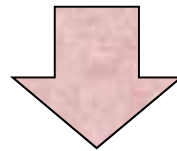
ACTION !

Cont... *T2 (for each statement)*

	Salary	DNO
	200	
	300	

EVENT: ONLY «SAL» changes and All DNOs are NULL !

Old O

Salary	DNO
200	Null
300	Null

New N

Salary	DNO
700	Null
900	Null
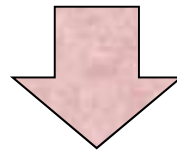
No action !

GÜVENLİK

- **Authentication** (**Kimlik doğrulama**), kullanıcının iddia ettiği kişi olduğunun ve DBMS'nin de kullanılmak istenen sistem olduğunun doğrulanması.
  - Kimlik doğrulama, belirli görevler için herhangi bir ayrıcalık/yetki vermez. **Authorization** (**Yetkilendirme**) için bir önkoşuldur.

- **Authorization (yetkilendirme);** hangi kullanıcının, hangi veriler üzerinde, neler yapabileceğinin, belirlenmesi.

- **Authentication «SERVER», Authorization «SERVER», Auditing «SERVER», …**

Authorization methods (yetkilendirme metodları)

- **Discretionary access control** *(takdire dayalı)*
  - Privileges are associated with <u>tables</u>
  - Based on the assumption that the users are trustworthy.
  - Privilege may propagate from one subject to another
  - Provided by SQL and flexible..
- **Mandotary access control** *(zorlayıcı)*
  - Privileges are associated with <u>data</u>
  - Eliminates the <u>loopholes</u> of dicreationary method, thus suited for high-security systems (like military)
  - Less flexible

Discretionary access control
Privileges (hak, yetki)

- The <u>table's owner</u> grants privileges on it to other users.

- Kinds of privileges:
  - *Select, Insert, Delete, Update*, *References, Usage, Trigger, Execute, Under*

- SQL **GRANT/REVOKE** *statement* assigns/removes priveleges to users:

  SYNTAX : **GRANT** *<priv list>* **ON** <DB element> **TO** *<users/roles>*

  Example: GRANT *select* on COURSE to public

UNIVERSITY database

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```

```
grant select on STUDENT to dean, admissions
grant insert on STUDENT to admissions
grant delete on STUDENT to dean
grant update on STUDENT to dean

grant select on COURSE to public
grant insert on COURSE to registrar
grant delete on COURSE to registrar
grant update on COURSE to registrar

grant select on DEPT   to public

grant select on ENROLL to dean, professor
grant insert on ENROLL to registrar
grant delete on ENROLL to registrar
grant update on ENROLL to professor

grant select on SECTION to public
grant insert on SECTION to registrar
grant delete on SECTION to registrar
grant update on SECTION to registrar
```

Figure 5-6
Some SQL grant statements for the university database

22

users / roles

- A "user" is the person who logged into the database. The "role" is the category of users.

- For a better feasibility, privileges are granted to "roles". Thus the admin should
  - define "roles" in the database.
  - define "users" in the database.
  - assign <u>one or more</u> roles to the user. *(In case of more than one role, the user is granted the union of priveleges from each role)*

- The owner of an db-object (table, *view, idx*,..) automatically gets all previliges on that object.

Fine-tuning: Column privileges

GRANT select on ENROLL to dean, professor

- The purpose of this privilege was for keeping student enrollments and grades private.

- Instead of "all or nothing" aspect, it is better to use column priveleges.

- Study the following column privileges:

GRANT *select(StudentId,grade)* on ENROLL to dean, professor

GRANT *update(Grade)* on ENROLL to professor

GRANT *insert(Sname,MajorId*) on STUDENT to admissions

Privileges required by SQL statements:

- **Select** requires "*select privilege*" on every field mentioned in the query.

- **Insert (update)** requires "*Insert (update)* privilege" for the field to be inserted (modified), **plus the "select privilege" on every field mentioned in its where clause.**

- **delete** requires "*delete privilege*" for the field to be deleted, plus the "select privilege" on every field mentioned in its where clause.

Statement	Required Privileges
select s.SId, s.SName, count(e.EId) from STUDENT s, ENROLL e where s.SId = e.StudentId and e.Grade = 'A' group by s.SId, s.SName	select(SId, SName) on STUDENT select(EId, StudentId, Grade) on ENROLL
select c.* from COURSE c where c.DeptId in (select d.DId from Dept d where d.DName = 'math')	select on COURSE select(DId, DName) on DEPT
delete from SECTION where SectId not in (select e.SectionId from ENROLL e)	select(SectId) on SECTION delete on SECTION select(SectionId) on ENROLL

Example: Determine Who is authorized to execute the SQL statements in the table.

GRANT *update(Grade)* on ENROLL to professor

Can a professor do the followings ?
UPDATE ENROLL e SET e.Grade= 100
UPDATE ENROLL e SET e.Grade=e.Grade+1

25

More Fine-tuning: Privileges on **Views**

GRANT *update(Grade*) on ENROLL to professor

- Column privileges gives roles(users) too much power. A professor can change a grade given by another professor.
- Solution: Revoke the previous privileges of professors on ENROLL table. Give the privileges on the following view.

```
create view PROF_ENROLLMENTS as
    select e.*
    from  ENROLL  e
    where  e.SectionId in
        (select k.SectId
          from Section k
          where k.Prof=current_user)


grant select        on  PROF_ENROLLMENTS  to professor
grant update(Grade)  on  PROF_ENROLLMENTS  to professor
```

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```

Create view **CurrYearGrades** AS
S YearOffered, Grades
F ENROLL e, SECTION s
W e.SectionId=s.SectId and YearOffered = extact(YEAR,current_date);

GRANT *select* ON **CurrYearGrades** to DEAN

Privileges on constraints

- "assertions" restricts the contents, limits what the user can do.

- It is not difficult for a malicious user to define an assertion that would make a table useless.

create assertion MakeTableUseless

check (not exist

select s.*

from STUDENT

where SName !='abc' and GradYear != 0)

- Thus, allowing to define assertions should be under authorization. We should control to use constraints (assertions, **Foreign keys**,...) that mentions STUDENT table.

GRANT *references* on STUDENT to dean

Reference vs. *Select* Privilege

```
STUDENT(SId, SName, GradYear, MajorId)
                FK

ENROLL(EId, StudentId, SectionId, Grade)
```
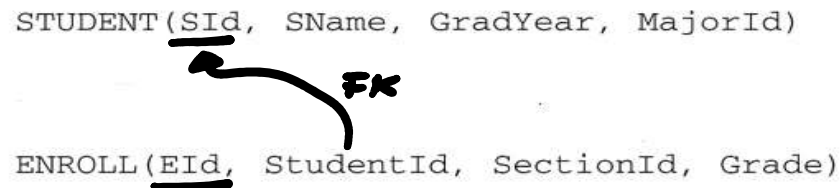
User U: CREATE TABLE STUDENT (…..)

U: GRANT **SELECT (SId**) on STUDENT to **V**

V: CREATE TABLE ENROLL (…….PK (EId), **FK(StudentId) REFERENCES STUDENT**, FK(SectionId) ….) ➔ **REJECTED !!**

U: GRANT **REFERENCES(SId**) on STUDENT to **V**

V: CREATE TABLE ENROLL (…….PK (EId), **FK(StudentId) REFERENCES STUDENT**, FK(SectionId) ….) ➔ **ACCEPTED!!**

Why SELECT privilege is not ENOUGH ?

Remember FK enforcements: (CASCADE, SET NULL, SET DEFAULT, **NO ACTION**)

V: CREATE TABLE ENROLL (…….PK (EId), FK(StudentId) REFERENCES STUDENT ON DELETE **NO ACTION**, FK(SectionId) ….)

Grant-option Privileges

GRANT *<priv list>* **ON** <DB element> **TO** *<users/roles>*

**GRANT** insert **ON** student **TO** admission WITH GRANT OPTİON

- The above grant grants the following previliges to admission:
  - To insert records on STUDENT table
  - To grant that privilege to others.
- Since it is a powerful, it is critical to grant that kind of privileges..

GRANT Diagrams

- Useful to represent «sequence of GRANTs» by a GRAPH.

- SQL system keeps track of all users privileges with «access matrices», Access Control Lists (ACL).

user name

privilege

ownership. state ($*$ or $**$)
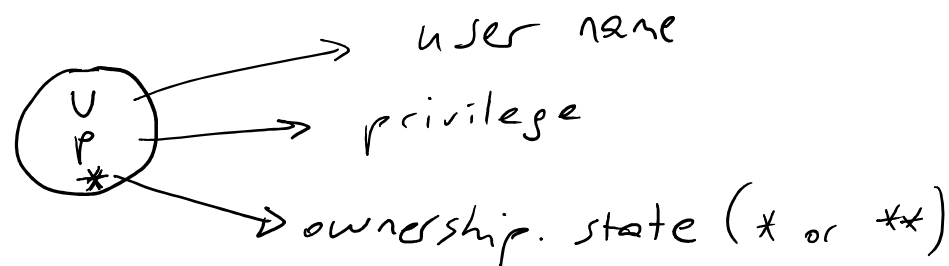
$\begin{cases} ** : \text{privilege is from ownership.} \\ * : \quad " \quad \text{is granted from} \\ \quad s/w \text{ else. It includes "with grant option"} \end{cases}$
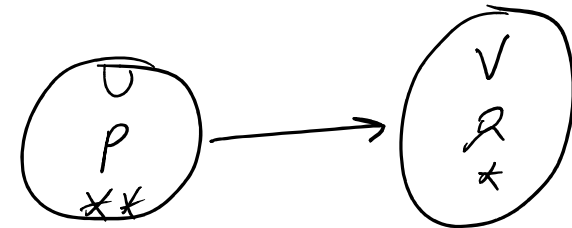
U grants Q to V
w/ grant option

$(Q \subset P)$

Example:

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```

User «**admin**» grants following privileges:

- GRANT *select,insert* ON ENROLL to prof,asistan WITH GRANT OPTION

User in «prof» role, grants following privileges:

- GRANT *select,insert* ON ENROLL to sekreter

User in «asistan» role, grants following privileges:

- GRANT *select,insert(EId*) ON ENROLL to sekreter
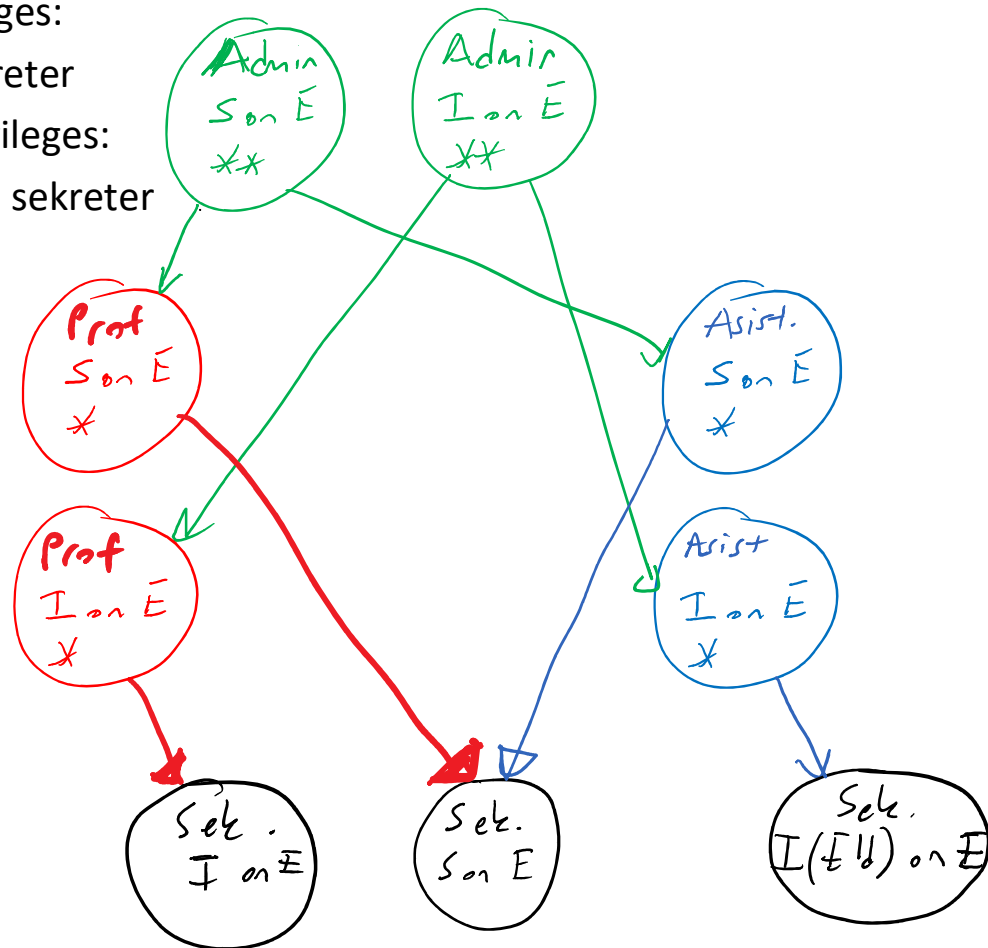
GRANT Diagrams/ REVOKE

REVOKE *<priv list>* **ON** <DB element> **FROM** *<users/roles>*
<CASCADE/RESTRICT>

CASCADE : specified privileges revoked **AND** privileges that were granted ONLY because of the granted priviliges. **At last, any node that is not accessible from some ownership node is also deleted.**

RESTRICT: If the privileges that are being tried to be revoked have been passed to others, then REJECT this present REVOKE.

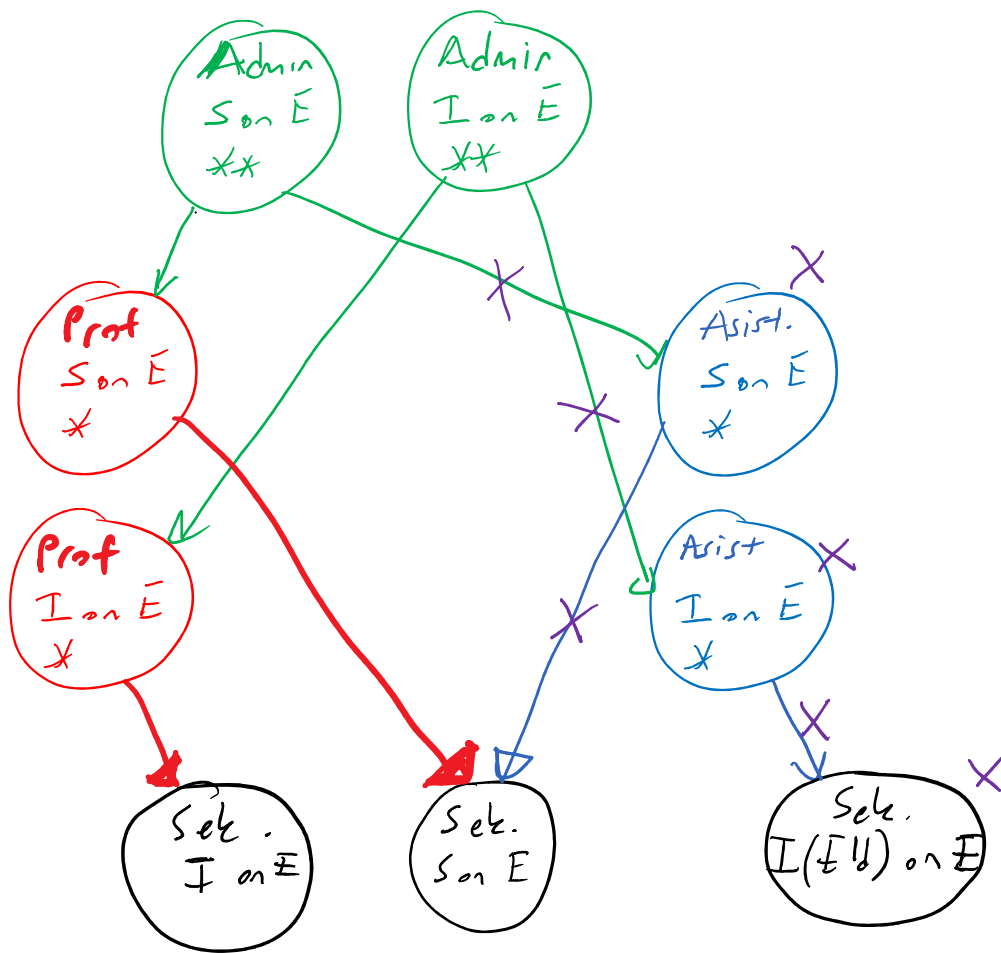**REVOKE** <u>**GRANT OPTION FOR**</u> <DB element> **FROM** *<users/roles>* **<CASCADE/RESTRICT>**
Privileges themselves remain but, the option to grant them is revoked.

Example:

```
STUDENT(SId, SName, GradYear, MajorId)
DEPT(DId, DName)
COURSE(CId, Title, DeptId)
SECTION(SectId, CourseId, Prof, YearOffered)
ENROLL(EId, StudentId, SectionId, Grade)
```
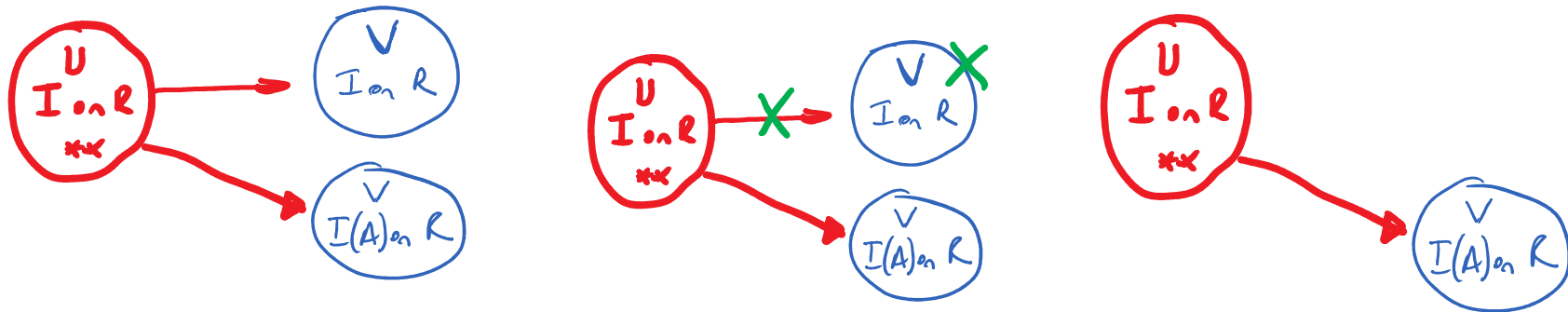
User «**admin**» grants following privileges:

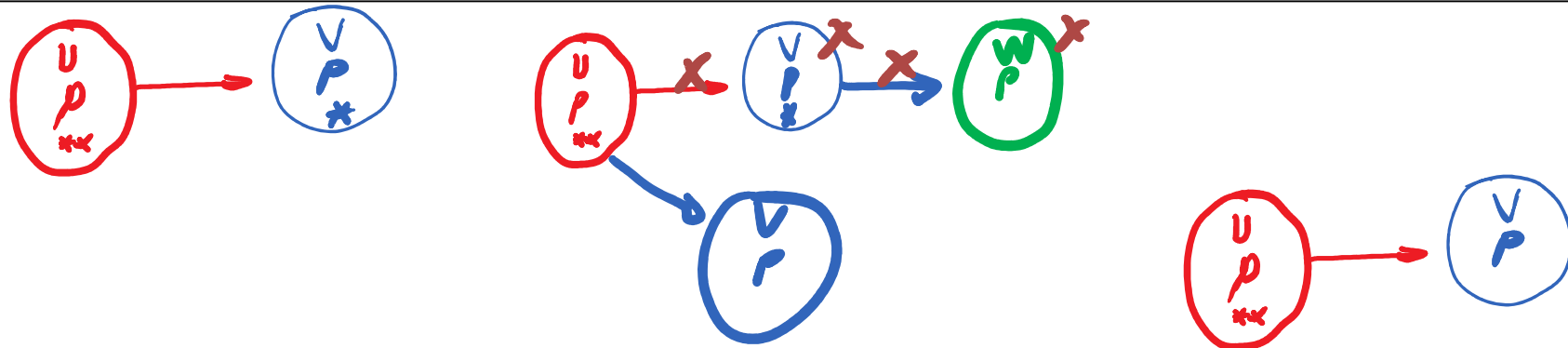- REVOKE *select,insert* ON ENROLL FROM asistan CASCADE
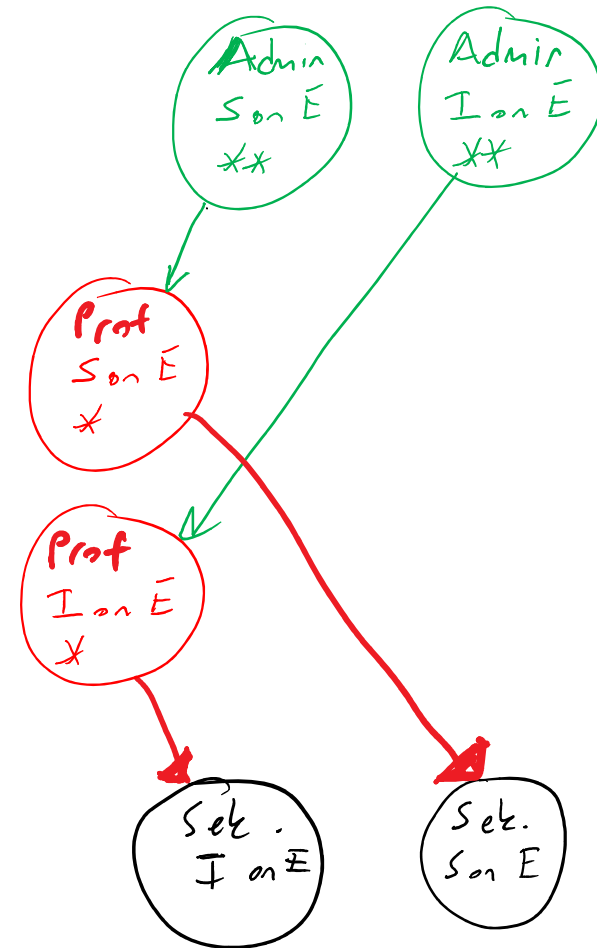
Example:

- U: GRANT *insert* ON R to V

- U: GRANT *insert(A)* ON R to V

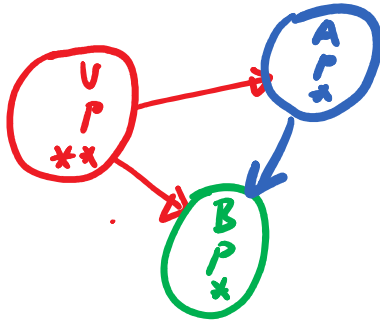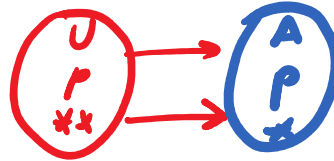- U: REVOKE *insert* ON R from V RESTRICT

Example:

- **U**: GRANT *p* to **V** WITH GRANT OPTION
- **V** : GRANT *p* to **W**
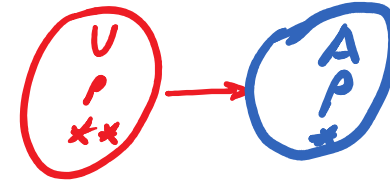- **U**: REVOKE **GRANT OPTION FOR** *p* from **V** CASCADE
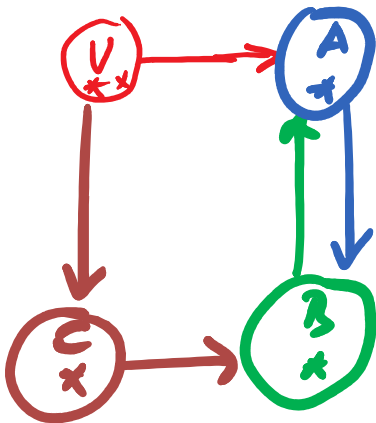
Examples:

U: REVOKE *p* FROM A CASCADE
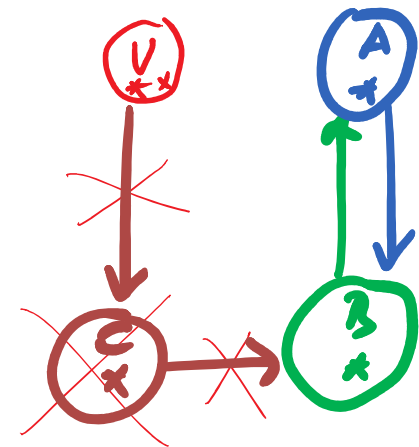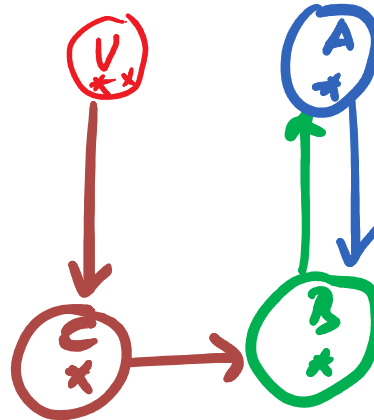
U: REVOKE *p* FROM A CASCADE

U: REVOKE **GRANT OPTION FOR** *p* FROM A CASCADE

U: REVOKE *p* FROM A CASCADE

U: REVOKE *p* FROM C CASCADE

Example

- A: GRANT *p* to B WITH GRANT OPTION
- A : GRANT *p* to C
- B : GRANT *p* to D WITH GRANT OPTION
- D : GRANT *p* to B, C, E WITH GRANT OPTION
- B : REVOKE *p* from D CASCADE
- A : REVOKE *p* from C CASCADE

Show the GRANT diagram steps..

Mandatory access control

- Remember that **discretionary controls** assume that **the users are trustworthy**.
- A LOOPHOLE scneario:
- A «malicious» professor can easily violate confidentiality.

```
create table TakeAPeek(StudentId int, Title varchar(20),
                       Grade varchar(2));

grant select on TakeAPeek to public;

insert into table TakeAPeek(StudentId, Title, Grade)
    select e.StudentId, c.Title, e.Grade
    from ENROLL e, SECTION k, COURSE c
    where e.SectionId=k.SectId and k.CourseId=c.CId;
```

- Since the professor is the owner, she can authorize anybody to look at "student grades" which is supposed to be private.
- Mandatory access control ensure confidentiality by assigning privileges to data, rather than to users.
- The IDEA in Mandatory access control : **When private data is copied to other tables, it still remains confidential.**
- This idea is implemented by **classification levels**. It may be built with security database software and additional discretionary SQL tools.

Security levels

- Each user,**S** and every object,**O** (table,record,…) and is assigned a clearances and security level.
    - Top secret **(TS)**
    - Secret **(S)**
    - Classified **(C)**
    - Unclassfied **(UC)**

- BELL-LaPADULA Model:

    - **Read Property: S** can **read O**; IFF **class(S) ≥ class(O):** A user is authorized to READ an object «only if» his level is at least as **HİGH** as the security level of the object.

    - **Write Property: S** can **write O**; IFF **class(S) ≤ class(O):** A user is authorized to WRITE an object «only if» his level is at least as **LOW** as the security level of the object.
    (Information MUST flow from lower security level to higher)

```
create table TakeAPeek(StudentId int, Title varchar(20),
                       Grade varchar(2));
insert into table TakeAPeek(StudentId, Title, Grade)
   select e.StudentId, c.Title, e.Grade
   from ENROLL e, SECTION k, COURSE c
   where e.SectionId=k.SectId and k.CourseId=c.CId;
```

- class(Prof) = S, class(TakeAPeek)= S
- Other users (like students,..) class(student) = C.   They cannot READ due to Read prop.

Examples:

- **A LOOPHOLE scneario:**

  User «Tricky» : CREATE TABLE T (….)

  «Tricky» : GRANT *insert* on **T** to Prof;

  Tricky modify app code of User «Prof» that enters Student Grades by writing(copying) grades into table **T.**

  «Prof» enters grades that is supposed to be secret. However, it is seen by «Tricky».

- <u>In case of Mandatory Model:</u>

  class(Prof) = S ➔ class(Application)=S

  class(Tricky) = C ➔ class(T) = C

  Due to Write prop. Prof's application CANNOT write to T.

```
select  c.Title
from  ENROLL  e,  SECTION  k,  COURSE  c
where  e.SectionId=k.SectId  and  k.CourseId=c.CId
and  e.Grade='F'
```

- Suppose ENROLL's level is *secret*, SECTION's level is *classified* and COURSE's level is *unclassified*. In such a case,
  - the user that has at least a level of *secret* is able to execute the query.
  - The **query's output table** has **the highest level** of its input tables = **S**

Covert Channels

- **«Inference» of information** about DB objects
  - that have higher security levels. (mandatory model)
  - in statistical DBs. (s.a. OLAP, Data warehouse)
  - Distributed databases. (more complex scenarios..)

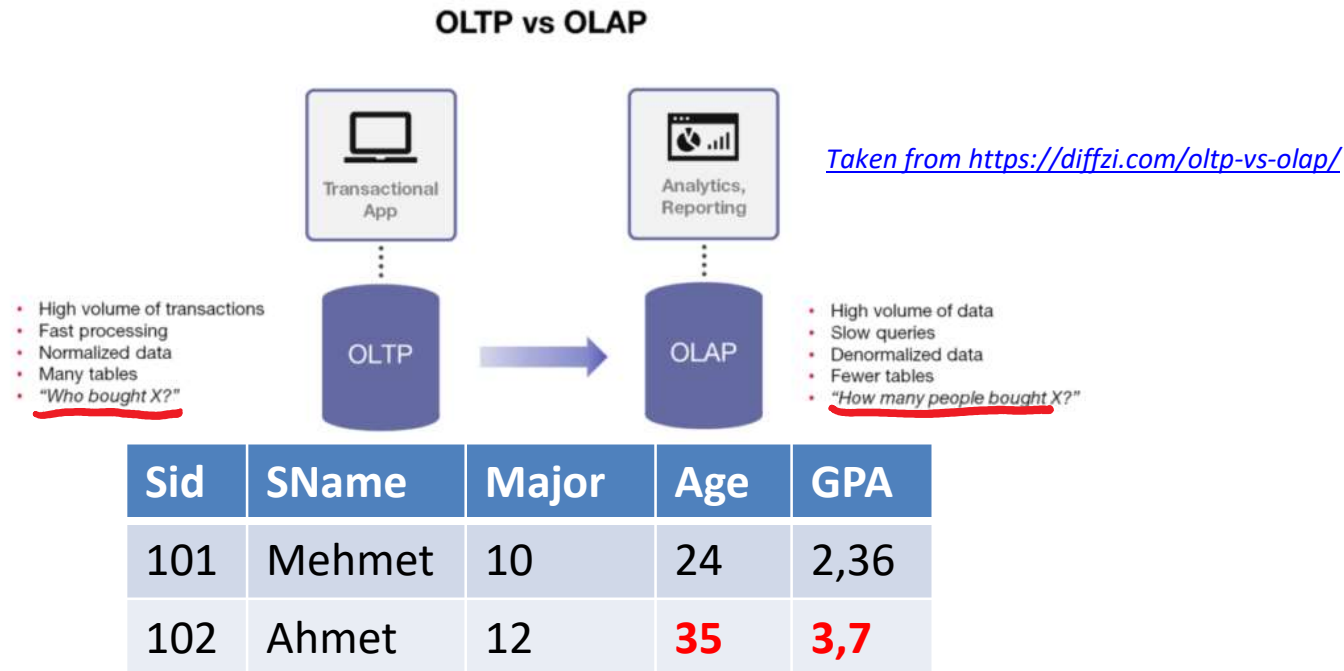| Eid	SectionId	StudentId	Grade	Security Class
101	10	1234	A	**S**
102	11	1235	B	**C**

- Users with **U** clearance can see 0 tuple.
- Users with **S** or **TS** clearance can see 2 tuple.
- Users with **C** clearance can see 1 tuple.

Scenario: Users with **C** clearance: **insert <101, …..>**

    If Accept: PK violation !

    If Reject: Covert Channel !

Statistical DBs (OLAP)

OLTP vs OLAP

Transactional App		Analytics, Reporting

- High volume of transactions
- Fast processing
- Normalized data
- Many tables
- *"Who bought X?"*

OLTP → OLAP

- High volume of data
- Slow queries
- Denormalized data
- Fewer tables
- *"How many people bought X?"*

Sid	SName	Major	Age	GPA
101	Mehmet	10	24	2,36
102	Ahmet	12	**35**	**3,7**

- OLAP can allow **only** statistical queries.
  - Everyone knows Ahmet is the <u>oldest</u> <u>student</u>. Can «User-Tricky» know Ahmet's age, GPA ..?
  - Tricky <u>repeatedly</u> ask : **Q1: «How many** students are there whose **age** is greater_equal **(≥)** than X?» **(increment X until the answer is 1)** ➔ **X=35 !!**
  - Tricky ask : **Q2:** «What is the **max** GPA of students whose **age** is greater_equal **(≥)** than **X=35?**» ➔ Tricky gets Ahmet's GPA=3,7

Preventing inference in OLAP

- «Each query MUST involve at least **N** rows to run»
  - Q1 runs. But Q2 fails. Tricky is decisive, does not give up :)..
  - Tricky <u>repeatedly</u> ask : **Q1: «How many** students are there whose **age** is greater_equal **(≥)** than X?» **(increment X until Result is N) , Say, X=28**
  - **Q3:** «What is the **sum of** GPA of all students whose **age** is greater_equal **(≥)** than X=28?» ➜ **The result includes «Ahmet»'s GPA.**
  - **Q4:** «What is the **sum of** GPA of all students **other than «Ahmet»,** whose age is greater_equal **(≥)** than X=28 and **including «himself(Tricky)?»**
  - **Ahmet's GPA= Q3- (Q4-Tricky's own GPA)** ➜ Tricky gets Ahmet's GPA=3,7

SUMMARY on Authorization methods

Prefered in commercial databases

- Discretionary access control
  - Privileges are associated with <u>tables</u>
  - Based on the assumption that the users are trustworthy.
  - Provided by SQL and flexible
- Mandotary access control
  - Privileges are associated with <u>data</u>
  - Eliminates the loopholes of dicreationary method, thus suited for high-security systems (like military)
  - Less flexible