Universiteit Antwerpen

Introduction to Databases
Lecture 6:
Active Databases

Floris Geerts

Universiteit Antwerpen

Today's Menu

- Constraints
  - Table-level constraints
    - UNIQUE, NOT NULL, PRIMARY KEY, FOREIGN KEY (repetition)
    - CHECK
    - DOMAIN
  - Schema-level constraints
    - Assertions
  - Deferred versus Immediate
  - Modifying Constraints
- Reacting to Actions
  - ON DELETE CASCADE
  - Triggers

Relation-Level Constraints

PRIMARY KEY & UNIQUE

- Defines a set of attributes that together uniquely identify a tuple in a relation.

FOREIGN KEY

- Defines a set of attributes that uniquely identify a tuple from one relation to another relation.
- Can be employed to enforce referential integrity.

NOT NULL

- Disallows NULL values for attributes.

CHECK

- Allows for the specification of more flexible value- and tuple-level constraints.

Primary Keys

A **primary key** of a relation *R* is a set of attributes that together uniquely identify a tuple in *R*.

- The projection of the tuples in a relation instance on the primary key of its schema may not contain duplicates.
- Every relation schema may have *at most* 1 primary key.

- SQL has two options for defining the primary key of a relation:

```
CREATE TABLE MovieStar (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        gender CHAR(1),
        birthdate DATE );
```

This is only possible if the primary key contains exactly one attribute.

```
CREATE TABLE MovieStar (
        name CHAR(30),
        address VARCHAR(255),
        gender CHAR(1),
        birthdate DATE,
        PRIMARY KEY (name,address));
```

This allows any combination of attributes to be the primary key.

Primary Key vs. Unique

If, additionally, we want to express that on other projections, the relation instance may not contain duplicates, we use **UNIQUE**.
UNIQUE may be used similarly to how PRIMARY KEY is used, but there may be more than one UNIQUE constraint per relation.

```
CREATE TABLE MovieStar (
        name CHAR(30),
        address VARCHAR(255),
        gender CHAR(1),
        birthdate DATE,
        PRIMARY KEY(name),
        UNIQUE (address));
```

SQL standard: UNIQUE constraint only needs to hold for tuples not having NULL in any of the unique attributes. Not all DBMSs support this feature correctly.

We can only use unique column combinations in foreign key dependencies.
Primary key is always UNIQUE NOT NULL

Foreign Keys

A **foreign key** is a set of attributes in a relation *R* that together uniquely identify a tuple in another relation *S* referenced by *R*.

- The projection of the tuples in relation *R* onto a foreign key may contain duplicates. The referenced attributes must be declared UNIQUE or be the PRIMARY KEY in the relation schema of *S*.
- A relation schema may have multiple foreign keys.
- Each foreign-key constraint from a relation *R* to a relation *S* defines a N:1 relationship from *R* to *S*.
- Special case: if the set of attributes $A_1,...,A_n$ in *R* which reference *S* is a key of *R*, then we have a 1:1 relationship.

Defining Foreign Keys

```
CREATE TABLE MovieExec (
        name CHAR(30) UNIQUE,
        address VARCHAR(255),
        certN INT PRIMARY KEY,
        netWorth INT);
```

- SQL has two options for defining a foreign key of a relation:
  - Only one attribute forms the FK:

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT REFERENCES MovieExec(certN));
```

  - Allows for multiple attributes to form together the FK:

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT,
        FOREIGN KEY (presCertN,name) REFERENCES MovieExec(certN,name));
```

NOT NULL Constraints

If we do not want to allow foreign keys to be NULL, we can express this by a NOT NULL constraint. (This effectively turns a foreign key constraint into a true referential integrity constraint in the ERM world.)

- A PRIMARY KEY may never be NULL; a UNIQUE constraint may.

```
CREATE TABLE MovieStar (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        gender CHAR(1) NOT NULL,
        birthdate DATE);

CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT NOT NULL REFERENCES MovieExec(certN));
```

Today's Menu

- Constraints
  - Table-level constraints
    - UNIQUE, NOT NULL, PRIMARY KEY, FOREIGN KEY (repetition)
    - CHECK
    - DOMAIN
  - Schema-level constraints
    - Assertions
  - Deferred versus Immediate
  - Modifying Constraints
- Reacting to Actions
  - ON DELETE CASCADE
  - Triggers

Value-based CHECK

CHECK constraints allow us to express additional value restrictions for attributes (not only within keys)

```
CREATE TABLE MovieStar (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        gender CHAR(1) CHECK (gender IN ('M',’F’,’X’)),
        birthdate DATE );

CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT CHECK (presCertN >= 1000000) REFERENCES MovieExec(certN) );
```

Tuple-based CHECK

```
CREATE TABLE MovieStar (
        name CHAR(30) UNIQUE,
        address VARCHAR(255),
        gender CHAR(1),
        birthdate DATE,
        CHECK (gender = 'F' OR name LIKE ‘An.%'));
```

- Here, the GENDER and NAME attributes refer to the *MovieStar* relation;
- the condition can be verified tuple by tuple;
- the condition is only verified when the table is updated or a tuple is inserted to the table;

Subqueries within CHECK Constraints

(not allowed in Oracle nor in Postgres)

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255)
                CHECK (address NOT IN
                        (SELECT address
                        FROM MovieStar),
        presCertN INT REFERENCES MovieExec(certN) );
```

- Here, the ADDRESS attribute of the CHECK refers to the *Studio* relation.

Subqueries within CHECK Constraints

(not allowed in Oracle nor in Postgres)

```
CREATE TABLE R (
        a CHAR(10) PRIMARY KEY,
        b CHAR(10) CHECK (EXISTS (SELECT * FROM S WHERE b=S.a)));
```

OK

```
CREATE TABLE R (
        a CHAR(10) PRIMARY KEY,
        b CHAR(10) CHECK (b IN (SELECT a FROM S)));
```

OK

```
CREATE TABLE R (
        a CHAR(10) PRIMARY KEY,
        b CHAR(10),
        CHECK ((b, a) IN (SELECT a,b FROM S)));
```

EVEN THIS IS OK!

Evaluation of CHECK Constraints

CHECK constraints are only verified when the related attributes are updated or a tuple is inserted to the table on which the CHECK is defined.

- This means that the CHECK condition can be violated without changing the value of the attribute, but by changing something that influences the attached subquery.

- Here is an erroneous simulation of referential integrity:

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT NOT NULL REFERENCES MovieExec(certN));
```

← **CORRECT**

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT NOT NULL CHECK (presCertN  IN (SELECT certN FROM MovieExec));
```

← **NOT CORRECT**

Domain Constraints

(not allowed in Oracle, CHECK with subqueries not allowed in Postgres)

Create a domain with a CHECK constraint, use the domain afterwards.

```
CREATE DOMAIN GenderDomain CHAR(1) CHECK (VALUE IN ('M',’F’,’X’));

CREATE TABLE MovieStar (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        gender GenderDomain,
        birthdate DATE);


CREATE DOMAIN StudioAddrDomain VARCHAR(255)
                CHECK (VALUE NOT IN (SELECT address FROM MovieStar));

CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address StudioAddrDomain,
        presCertN INT REFERENCES MovieExec(certN) );
```

Today's Menu

- Constraints
  - Table-level constraints
    - UNIQUE, NOT NULL, PRIMARY KEY, FOREIGN KEY (repetition)
    - CHECK
    - DOMAIN
  - Schema-level constraints
    - Assertions
  - Deferred versus Immediate
  - Modifying Constraints
- Reacting to Actions
  - ON DELETE CASCADE
  - Triggers

Schema-Level Constraints

(not supported in Oracle/Postgres, uses CREATE TRIGGER instead)

ASSERTION

- Allows for the specification of schema-level (i.e., database-wide) consistency constraints. These are checked upon *every update (insert-delete-update)* to the database.

CREATE ASSERTION <name> CHECK (<condition>) ;

- where all attributes in the <condition> have to refer to a SELECT-FROM-WHERE statement.
- The condition will be checked every time when there is a deletion, insertion or update in the entire database schema!

Assertions

(not supported in Oracle/Postgres, uses CREATE TRIGGER instead)

MovieExec(name, address, certN, netWorth)
Studio(name, address, presCertN)

- No one may be the president of a studio, unless this person has a net worth of at least $1M :

```
CREATE ASSERTION RichPres CHECK
        (NOT EXISTS (
                SELECT * FROM MovieExec, Studio
                WHERE certN = presCertN AND netWorth < 1000000));
```

- Every executive name has only one address :

```
CREATE ASSERTION FunctionalDependency CHECK
        (NOT EXISTS (
                SELECT * FROM MovieExec mv1, MovieExec mv2
                WHERE mv1.name = mv2.name AND mv1.address <> mv2.address));
```

More ASSERTION Examples...

Movie(title, year, length, inColor, studioName, producerCertN)

The total length of all the movies by a given studio shall not exceed 1000000 minutes:

```
CREATE ASSERTION SumLength CHECK (
        1000000 <= ALL (
                SELECT SUM(length) FROM Movie GROUP BY studioName));
```

Two equivalent constraints:

```
CREATE TABLE R(a INT CHECK (a < 10));
```

is equivalent to

```
CREATE TABLE R(a INT);

CREATE ASSERTION ValueCheck CHECK (10 > ALL (SELECT a FROM R));
```

More ASSERTION Examples...

Referential integrity as an ASSERTION:

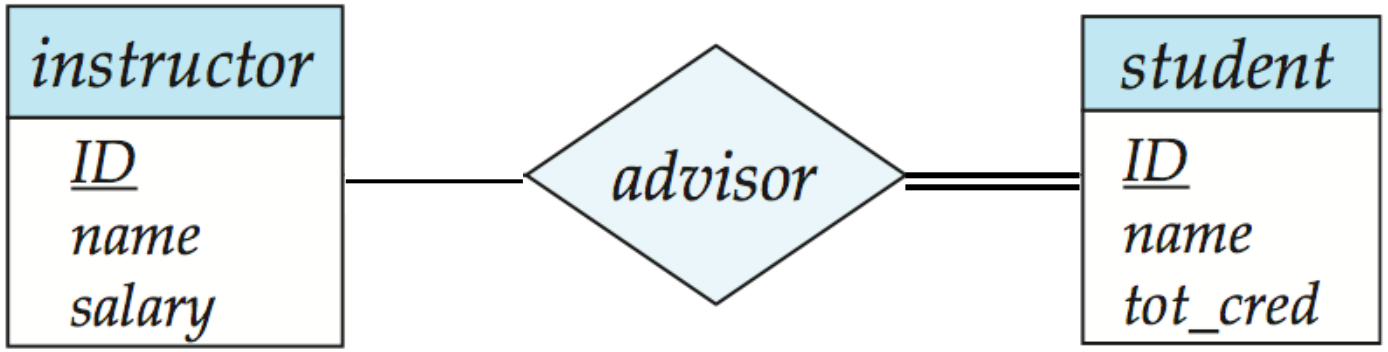
```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT,
        FOREIGN KEY presCertN NOT NULL REFERENCES MovieExec(certN));
```

is equivalent to:

```
CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT);

CREATE ASSERTION RefIntAssertion CHECK (
        NOT EXISTS (    (SELECT presCertN FROM STUDIO WHERE presCertN IS NULL)
                        UNION (  (SELECT presCertN FROM STUDIO)
                                 EXCEPT (SELECT certN FROM MovieExec))));
```

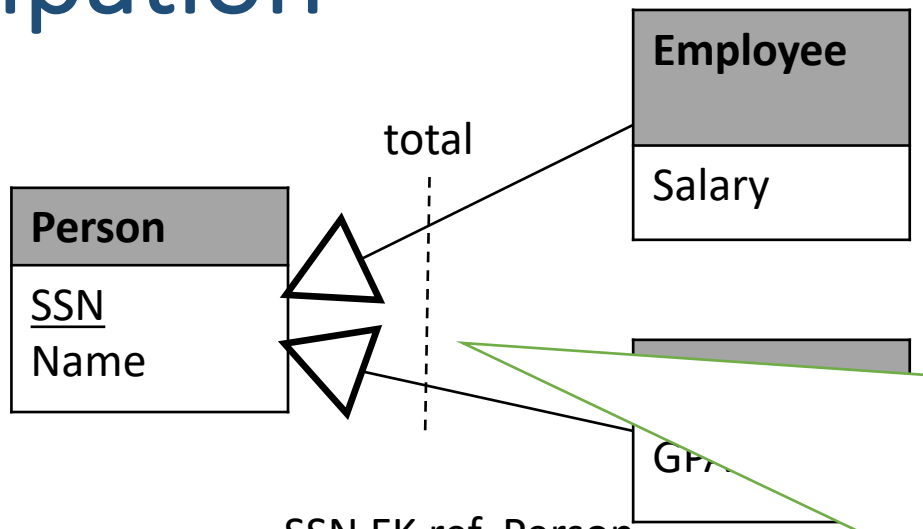
CHECK and ER diagrams



instructor(iID, name, salary)
student(sID,name,tot_cred)
advisor(iID,sID,date) iID FK ref. instructor ; sID FK ref. student

```
CHECK ASSERTION StudentIsAdvised
 CHECK ( NOT EXISTS (SELECT sID FROM student
                     WHERE NOT EXISTS ( SELECT * FROM advisor as A
                                        WHERE A.Sid=siD));
```

Total Participation



- Person(SSN,Name)
- Employee(SSN,salary) SSN FK ref. Person
- Student(SSN,GPA) SSN FK ref. Person

Need to express: every person is either an employee or a student.
This is equivalent to:
There **cannot** be a person that is **neither** an employee **nor** a student
This is equivalent to:
Not exists person which satisfies the conjunction
Person is **not** an employee
and
Person is **not** a student
This is equivalent to
Not exists person which does **not** satisfy the disjunction
Person is student or Person is employee

```
CHECK ASSERTION PersonisEorS
  CHECK ( NOT EXISTS (SELECT SNN FROM Person as P
                      WHERE NOT EXISTS ( SELECT * FROM Employee as E WHERE E.SNN=P.SNN) AND
                            NOT EXISTS  ( SELECT * FROM Student as S  WHERE S.SNN=P.SNN) ));
CHECK ASSERTION PersonisEorS
  CHECK ( NOT EXISTS (SELECT SNN FROM Person as P
                      WHERE NOT EXISTS ( SELECT * FROM Employee as E WHERE E.SNN=P.SNN) OR
                            ( SELECT * FROM Student as S  WHERE S.SNN=P.SNN)) ));
```

Universiteit Antwerpen

Today's Menu

- Constraints
  - Table-level constraints
    - UNIQUE, NOT NULL, PRIMARY KEY, FOREIGN KEY (repetition)
    - CHECK
    - DOMAIN
  - Schema-level constraints
    - Assertions
  - Deferred versus Immediate (need: cyclic foreign keys, transaction)
  - Modifying Constraints
- Reacting to Actions
  - ON DELETE CASCADE
  - Triggers

Deferred vs Immediate

- But when are constraints checked?
  - At time of activity = IMMEDIATE
  - At the end of a transaction = DEFERRED
- A constraint that *can* be delayed until commit = DEFERRABLE
  - NOT NULL is always IMMEDIATE
  - Other constraints are by default NOT DEFERRABLE
- *Inside a transaction* a DEFERRABLE constraint can be set to DEFERRED:

  SET CONSTRAINTS [ALL|Constraint Name] DEFERRED
- Initial setting: INITIALLY [IMMEDIATE|DEFERRED]

Deferred vs Immediate: Example

```
CREATE TABLE MovieExec (
        name CHAR(30) UNIQUE,
        address VARCHAR(255),
        certN INT PRIMARY KEY,
        netWorth INT);

CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        FOREIGN KEY (presCertN) REFERENCES MovieExec(certN));

insert into Studio values ('Universal Studios','USA',10);
```

ERROR: insert or update on table "studio" violates foreign key constraint "studio_prescertn_fkey"
DETAIL: Key (prescertn)=(10) is not present in table "movieexec".

Deferred vs Immediate: Example

```
CREATE TABLE MovieExec (
        name CHAR(30) UNIQUE,
        address VARCHAR(255),
        certN INT PRIMARY KEY,
        netWorth INT);

CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT REFERENCES MovieExec(certN));

BEGIN;
insert into Studio values ('Universal Studios','USA',10);
insert into MovieExec values ('Steven','USA',10,10000000);
COMMIT;
```

ERROR: insert or update on table "studio" violates foreign key constraint "studio_prescertn_fkey"
DETAIL: Key (prescertn)=(10) is not present in table "movieexec".

Deferred vs Immediate: Example

```
CREATE TABLE MovieExec (
        name CHAR(30) UNIQUE,
        address VARCHAR(255),
        certN INT PRIMARY KEY,
        netWorth INT);


CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT CONSTRAINT studio_prescertn_fkey REFERENCES MovieExec(certN) DEFERRABLE) ;

BEGIN;
SET CONSTRAINTS studio_prescertn_fkey DEFERRED;
insert into Studio values ('Universal Studios','USA',10);
insert into MovieExec values ('Steven','USA',10,10000000);
COMMIT;
```

OK!

Deferred vs Immediate: Example

```
CREATE TABLE MovieExec (
        name CHAR(30) UNIQUE,
        address VARCHAR(255),
        certN INT PRIMARY KEY,
        netWorth INT);

CREATE TABLE Studio (
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presCertN INT REFERENCES MovieExec(certN) DEFERRABLE INITIALLY DEFERRED) ;

BEGIN;
insert into Studio values ('Universal Studios','USA',10);
insert into MovieExec values (‘Steven’,’USA’,10,10000000);
COMMIT;
```

OK!

Today's Menu

- Constraints
  - Table-level constraints
    - UNIQUE, NOT NULL, PRIMARY KEY, FOREIGN KEY (repetition)
    - CHECK
    - DOMAIN
  - Schema-level constraints
    - Assertions
  - Deferred versus Immediate
  - Modifying Constraints
- Reacting to Actions
  - ON DELETE CASCADE
  - Triggers

Modification of Constraints

Every constraint (except assertions, which have their own name) can be given a name, by letting it be preceded by: CONSTRAINT <name>

```
CREATE TABLE MovieStar (
    name CHAR(30) CONSTRAINT Key PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE);

CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presCertN INT CONSTRAINT RefInt REFERENCES MovieExec(certN));

CREATE DOMAIN CertDomain INT CONSTRAINT DomConst CHECK(VALUE >= 1000000);

CREATE TABLE MovieStar (
    name CHAR(30) CONSTRAINT Key PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1),
    birthdate DATE,
    CONSTRAINT Feminine CHECK (gender = 'F' OR name LIKE 'Ms.%'));
```

Modification of Constraints

- We can drop a constraint:

```
ALTER TABLE MovieStar DROP CONSTRAINT Key;

ALTER TABLE Studio DROP CONSTRAINT RefInt;

ALTER DOMAIN CertDomain DROP CONSTRAINT DomConst;

DROP ASSERTION RichPres;
```

- We can add a new constraint:

```
ALTER TABLE MovieStar ADD CONSTRAINT Key PRIMARY KEY(name);

ALTER TABLE Studio ADD CONSTRAINT RefInt FOREIGN KEY presCertN
   REFERENCES MovieExec(certN);

ALTER DOMAIN CertDomain ADD CONSTRAINT DomConst
  CHECK (VALUE >= 1000000) ;

ALTER TABLE MovieStar ADD CONSTRAINT Feminine
  CHECK (gender = 'F' OR name LIKE 'Ms.%');

CREATE ASSERTION RichPres CHECK ...;
```

ADD constraint checks the constraint and fails if the table does not satisfy the constraint.

Today's Menu

- Constraints
  - Table-level constraints
    - UNIQUE, NOT NULL, PRIMARY KEY, FOREIGN KEY (repetition)
    - CHECK
    - DOMAIN
  - Schema-level constraints
    - Assertions
  - Deferred versus Immediate
  - Modifying Constraints
- Reacting to Actions
  - ON DELETE CASCADE
  - Triggers

Reacting to Actions: Triggers and Cascades

- Mechanisms to maintain integrity in case of certain events
  - Deletion, insertion, update
- What to do with referential integrity:
  - Delete of referred tuple
  - Update of key value of referred tuple

- More general mechanism: Triggers
  - Whenever an event occurs, execute an activity
  - Event: insert/delete/update happened/is going to happen

3 Policies for Maintaining Referential Integrity

```
CREATE TABLE MovieExec (
    name CHAR(30) UNIQUE,
    address VARCHAR(255),
    certN INT PRIMARY KEY,
    netWorth INT);
```

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presCertN INT REFERENCES MovieExec(certN));
```

1. Default: **reject** any of the following modification attempts:
   - **Insert** a tuple into *Studio* whose presCertN value is not NULL and is not in the certN attribute of any *MovieExec* tuple;
   - **update** a tuple in *Studio* whose presCertN value is not NULL and is not in the certN attribute of any *MovieExec* tuple;
   - **delete** a tuple in *MovieExec* whose certN value appears in the presCertN attribute of some *Studio* tuple;
   - **update** a tuple in *MovieExec* by changing its certN value and whose (old) certN value appears in the presCertN attribute of some *Studio* tuple.

3 Policies for Maintaining Referential Integrity

```
CREATE TABLE MovieExec (
    name CHAR(30) UNIQUE,
    address VARCHAR(255),
    certN INT PRIMARY KEY,
    netWorth INT);
```

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presCertN INT REFERENCES MovieExec(certN) ON DELETE SET NULL);
```

2. **Set NULL** ON DELETE SET NULL – ON UPDATE SET NULL

- **Deleting** a tuple in *MovieExec*, whose certN value appears in the presCertN attribute of some *Studio* tuple, sets the presCertN value of the latter tuples to NULL;
- **updating** a tuple in *MovieExec* by changing its certN value, whose (old) certN value appears in the presCertN attribute of some *Studio* tuple, sets the presCertN values of the latter tuples to NULL.

3 Policies for Maintaining Referential Integrity

```
CREATE TABLE MovieExec (
    name CHAR(30) UNIQUE,
    address VARCHAR(255),
    certN INT PRIMARY KEY,
    netWorth INT);
```

```
CREATE TABLE Studio (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    presCertN INT REFERENCES MovieExec(certN) ON DELETE CASCADE);
```

3. **CASCADE**: ON DELETE CASCADE – ON UPDATE CASCADE

- **Deleting** a tuple in *MovieExec,* whose certN value appears in the presCertN attribute of some *Studio* tuple, leads to a deletion of the latter tuples;
- **Updating** a tuple in *MovieExec* by changing its certN value (whose (old) certN value appears in the presCertN attribute of some *Studio* tuple) leads to an update of the presCertN values of latter tuples to the new certN value.

These conditions apply recursively to all foreign-key constraints in the entire database schema!

Triggers (NOT PART OF EXAMINABLE MATERIAL)

TRIGGER

- Allows for the specification of schema-level (i.e., database-wide), event-driven consistency constraints. These are checked *before or after an event* that fires the trigger.
- More expressive and powerful than assertions.

Triggers in SQL

General syntax:

```
CREATE TRIGGER <trigger_name>
 {BEFORE|AFTER}  {{INSERT|DELETE|UPDATE} [OF <row_list>]}+  ON <table_name>
 [REFERENCING  [NEW ROW|TABLE AS <new_name>] [OLD ROW|TABLE AS <old_name>]]
 [FOR EACH ROW|STATEMENT [WHEN (<trigger_condition>)]]
BEGIN
 <trigger_body>
END;
```

* SQL'99 allows both ROW and TABLE/STATEMENT level triggers.
Oracle allows only ROW level, hence the keyword ROW is omitted.

Specifying Events in Triggers

- Triggering events may be of the form:
  - ... INSERT ON R ...
  - ... INSERT OR DELETE OR UPDATE ON R ...
  - ... UPDATE OF A, B OR INSERT ON R ...
- In SQL, if FOR EACH ROW option is specified, the trigger is row-level; otherwise, the trigger is statement-level.
  - The special variables NEW and OLD are available to refer to new and old tuples/tables, respectively.
- A trigger restriction can be specified in the WHEN clause, enclosed by parentheses. The trigger restriction is an SQL condition that must be satisfied in order to fire the trigger.
  - This condition cannot contain subqueries.
  - Without the WHEN clause, the trigger is fired for each row or the entire statement.

TRIGGER Example...

```
CREATE TABLE T4 (a INTEGER, b CHAR(10));
CREATE TABLE T5 (c CHAR(10), d INTEGER);
```

- *<trigger_body>* may be an entire sequence of SQL statements. However, one cannot modify the same relation whose modification is the event triggering the trigger
- The following trigger checks whenever a new tuple is inserted in T4, whether the new tuple has an a-value of 10 or less, and if so, it inserts the reverse tuple into *T5*:

```
CREATE TRIGGER trig1 AFTER INSERT ON T4
   REFERENCING NEW ROW AS NewRow
   FOR EACH ROW
   WHEN (NewRow.a <= 10)
   BEGIN
     INSERT INTO T5 VALUES(NewRow.b, NewRow.a);
   END;
```

Caution: Postgres only allows the execution of a user-defined function (UDF) as the trigger body.

INSTEAD-OF Triggers

- Not part of the SQL'99 standard but some systems (incl. Oracle, MS SQL Server, Postgres in combinination with a UDF) support this particular kind of triggers.

- INSTEAD OF indicates that the operation firing the trigger will not actually be executed, but another (sequence of) operation(s) specified in the trigger body will be executed *instead*.

Modifying Views with Triggers Example...

Movie(title, year, length, inColor, studioName, producerCertN)
MovieExec(name, address, certN, netWorth)

View definition:

```
CREATE VIEW MovieProd AS
  SELECT title, year, name, certN FROM Movie, MovieExec WHERE producerCertN = certN;
```

Trigger definition:

```
CREATE TRIGGER MovieProdInsert INSTEAD OF INSERT ON MovieProd
  REFERENCING NEW ROW AS NewRow FOR EACH ROW
  BEGIN
    INSERT INTO Movie(title, year, producerCertN)
       VALUES (NewRow.title, NewRow.year, NewRow.certN);
    INSERT INTO MovieExec(name, certN) VALUES (NewRow.name, NewRow.certN);
  END;
```

Triggers: Warning

- Triggers may interact and their behavior may become unpredictable
- One trigger inserts a tuple, triggering another trigger, which on its turn triggers ...
  - DBMS will execute triggers as long as there are new events
  - Provable hard problem to see if triggers are always well-behaved
- An incorrect trigger may cause inserts/updates/deletes to fail
  - Transaction ROLLBACK
- Triggers are not the way to go to implement business rules
  - Little flexibility
  - High complexity
  - Performance hit

Summary

- Table-level constraint: CHECK
- Schema-level constraint: ASSERTION
- Deferred versus Immediate

- Active Databases
  - Actions triggered to maintain integrity
  - ON DELETE CASCADE/ON UPDATE/….
  - **Triggers**