COMP353 Databases

More on SQL:

Null value

Triggers

Domains

- SQL allows user defined data types - **domains**
- We can define a domain as follows:
  - The keyword ***CREATE DOMAIN***
  - The name of the domain
  - The keyword ***AS***
  - Type description
  - Optional default value, constraints
- Example

  CREATE DOMAIN <name> **AS** <type description>

Domains

- To create a domain:
  - **CREATE DOMAIN** MovieDomain **AS VARCHAR(50);**
- Example:

```
CREATE TABLE Movie (
        title MovieDomain,
        year DATE,
        …….
        …….
);
```

Domains

- To create a domain with default value:
  - **CREATE DOMAIN** MovieDomain **AS VARCHAR(50)** **DEFAULT** ’unknown’**;**
- To change the default for a domain:
  - **ALTER DOMAIN** MovieDomain **SET DEFAULT** ’no such title’**;**
- To delete a domain definition:
  - **DROP DOMAIN** MovieDomain**;**

NULLs

- We use NULL in place of a value in a tuple's component when:
  - The value is *unknown* -- don't know
  - The value is *inapplicable* -- NA
  - The exact value *does not matter* -- don't care
- There could be many reasons why a value is not present in a relation, e.g., when *inserting* a tuple into a relation, we don't have/wish to specify all the values for the attributes.

Arithmetic operations on NULLs

- Result of an arithmetic operator, when at least one of the operands has a value of NULL, is NULL
- Example:
  - Suppose the value of attribute x is NULL

    ➔The value x+3 is also NULL
  - Note however that NULL is not a constant!

    ➔ NULL + 3 is ***illegal***

Arithmetic operations on NULLs

- Some key laws in math fail to hold with NULLs
- Suppose x is an attribute with numeric value
  - Example 1:
    - We know that $x * 0 = 0$, but
    - If x is NULL ➔ $x * 0$ is NULL
  - Example 2:
    - We also know that $x - x = 0$, but
    - If x is NULL ➔ $x - x$ is NULL

Comparison operations on NULLs

- The result of a comparison is “usually” **TRUE** or **FALSE**
- That is, 2 possible values or ***2-valued logic***
- Comparisons involving NULLs give rise to a **3rd** truth value, **UNKNOWN,** and hence we are dealing with a 3-valued logic
- In this case, the 3 possible values are **true, false, unknown**

3-Valued Logic

- We may assume that:
  - **TRUE** = 1
  - **FALSE** = 0
  - **UNKNOWN** = 1/2
- Then:
  - *x* **AND** *y* = min (*x*, *y*)
  - *x* **OR** *y* = max (*x*, *y*)
  - **NOT** *x* = 1–*x*

Truth table for 3-Valued Logic

X	*Y*	*X* AND *Y*	*X* OR *Y*	NOT *X*
TRUE	TRUE	TRUE	TRUE	FALSE
TRUE	UNKNOWN	UNKNOWN	TRUE	FALSE
TRUE	FALSE	FALSE	TRUE	FALSE
UNKNOWN	TRUE	UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE	UNKNOWN	UNKNOWN
FALSE	TRUE	FALSE	TRUE	TRUE
FALSE	UNKNOWN	FALSE	UNKNOWN	TRUE
FALSE	FALSE	FALSE	FALSE	TRUE

3-Valued Logic

- Some key laws in logic fail to hold with NULLs
- Example:
  - Law of the excluded middle
    - *x* **OR NOT** *x* = **TRUE**
  - For 3-valued logic
    - if *x* = **UNKNOWN** ➔ *x* **OR (NOT** *x***)** =

      max(1/2,(1−1/2)) = 1/2 = **UNKNOWN** ≠ (**TRUE)**
    - **Note: Do not treat NULL as a "value"**

      (e.g., see the next example)

Example

- Relation schema:

  Movie (title, year, length, filmType, studioName, producerC#)

- Consider query:

```
SELECT *
FROM Movie
WHERE length <= 120 OR length > 120;
```

- Don't we expect to get a copy of the **Movie** relation?
- Yes, if there is no Movie tuple whose length is NULL
- This query returns a subset of **Movie** tuples in general.
  It returns only each **Movie** tuple whose length is not NULL.

Example

- Read the box on page 254 in the textbook for some rules on NULLs
- The value NULL is ignored in any aggregation.
  - Query:

    ```
    SELECT COUNT(A) FROM R;
    ```

    returns the number of non-null values under attribute A in R.
  - Query:

    ```
    SELECT COUNT(*) FROM R;
    ```

    returns the number of tuples in R.
- NULL is treated as an ordinary value in a "group by" attribute.
  - Query:

    ```
    SELECT A, AVG(B) FROM R GROUP BY A;
    ```

    produces a tuple for each distinct value A, including the null, if exists.

Joins in SQL

- How to express the Cartesian product in SQL?
  - List the relation names in the **FROM** clause
- How do we express various joins in SQL?
  - Follow the Cartesian Product with conditions in the **WHERE** clause for the desired join

Joins in SQL2

- In SQL2, there are other forms for expressing $\times$ and $\rhd\lhd$
- Cartesian Product of Movie and StarsIn
  - Movie **CROSS JOIN** StarsIn; (Movie $\times$ StarsIn, in RA)
- Theta- (or equi-) join of Movie and StarsIn
  - Movie **JOIN** StarsIn **ON**
    StarsIn.title = Movie.title **AND** StarsIn.year = Movie.year;

  Note: The result may have some redundant columns;
  We can use the above expression as a subquery in a **FROM** clause and use **SELECT** to remove these undesired attributes
- Natural join of Movie and StarsIn
  - Movie **NATURAL JOIN** StarsIn;

Example

- Relation schemas:

  Exec(name, address, cert#, netWorth)

  Star (name, address, gender, birthdate)

- Query:

  Find all info on all stars who are also movie executives

- Query in SQL:

  SELECT *

  FROM Star **NATURAL JOIN** Exec;

  - The join expression appears in a **FROM** clause
  - Parenthesized **SELECT-FROM-WHERE** is also allowed in a **FROM** clause

Example

- Relation schemas:

  Exec(name, address, cert#, netWorth)

  Star (name, address, gender, birthdate)

- Query: Find "all" information on all stars
- Query in SQL:

  SELECT *

  FROM Star;

  - This query does not return all information on stars who are also movie executives
- Query in SQL

  SELECT *

  FROM Star **NATURAL JOIN** Exec ;

  - This query does NOT return stars who are not movie executives

Outer joins

- ***R* OUTER JOIN *S*** -- computes the join of ***R*** and ***S*** with **dangling** tuples padded with **NULL**s
- A tuple in ***R*** is **dangling** if it doesn't join with any tuple in ***S***
- Outer join could be
  - **FULL OUTER JOIN**
    - It pads, with nulls, the dangling tuples of ***R*** and ***S***
  - **LEFT OUTER JOIN**
    - It pads dangling tuples of ***R*** only
  - **RIGHT OUTER JOIN**
    - It pads dangling tuples of ***S*** only

Example

Instance R:

A	B
1	2
2	3

Instance S:

B	C
2	5
2	6
7	8

SELECT * FROM R NATURAL FULL OUTER JOIN S:

A	B	C
1	2	5
1	2	6
2	3	NULL
NULL	7	8

Example

Instance R:

A	B
1	2
2	3

Instance S:

B	C
2	5
2	6
7	8

R NATURAL LEFT OUTER JOIN S:

A	B	C
1	2	5
1	2	6
2	3	NULL

Example

Instance R:

A	B
1	2
2	3

Instance S:

B	C
2	5
2	6
7	8

R NATURAL RIGHT OUTER JOIN S:

A	B	C
1	2	5
1	2	6
NULL	7	8

Outer joins in SQL2

R **[NATURAL] [LEFT | RIGHT | FULL] OUTER JOIN** *S* **[ON ...]**

- Can do either a natural join ⟕ or a theta-join ⟕$_c$
- Use **NATURAL** for ⟕, and use **ON** for theta-outer join, but not both (NATURAL and ON cannot be used together)
- Can use any one of **LEFT**, **RIGHT**, or **FULL**

- Examples:
  - *R* **NATURAL FULL OUTER JOIN** *S*;
  - *Star* **NATURAL LEFT OUTER JOIN** *Exec*; (for slide #17)
  - Movie **RIGHT OUTER JOIN** StarsIn **ON** StarsIn.title = Movie.title **AND** StarsIn.year = Movie.year;

Constraints and Triggers

- SQL provides a variety of ways for expressing *integrity constraints* as part of the database schema
- Constraints' checking, in essence, provide users with more control over the database content
- An *active* element is a statement that we write once, store it in the database, and "expect" it to be executed at "appropriate" times
- The *time of action* might be when certain events occur (e.g., insertion of a tuple into a relation or any change(s) made to the database so that certain condition(s) becomes true

Constraints

- Declaration of primary keys
- Foreign key constraints (also see referential integrity constraints)
  - E.g., if in relation **StarsIn**, it says that a star has a role in a movie **m,** then there should be a movie tuple **m** in **Movie**
- Constraints on attributes, tuples, and relations
- SQL2 Assertions = global/general constraints (inter-relations)
  - Not supported in Oracle
- Triggers
  - are substitutes for “general” assertions
- SQL3 triggers and assertions

Primary Key

- A key constraint is declared by the DDL command **CREATE TABLE**
- Can use the keywords **PRIMARY KEY** or **UNIQUE**
  - Oracle treats them as synonyms
  - A table can only have one primary key but any number of "unique" declarations
- Two ways to declare a primary key in the **CREATE TABLE** statement:
  - After an attribute type, if the attribute is a key by itself
  - As a separate line, for any number of attributes forming the key:

    PRIMARY KEY (list of attribute(s))

    - **Should use the second way if the key is not a singleton**

Primary Key

- Example:

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

Three consequences of declaring a primary key:

1. Repeated values of the key attributes will not be allowed
   Any violation will be rejected/failed by the DBMS
2. NULLs are not allowed for the key attribute(s)
3. (Another possible consequence) Creating an index on the primary key, or alternatively keeping the table sorted on the key attribute(s).

Primary Key

- Example:

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

This is to say: ***no two movie stars have the same address***

Primary Key

- Example:

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

Note the distinction between **UNIQUE** and **PRIMARY KEY**: When using **UNIQUE**, NULLs are allowed for (all or some of) the key attribute(s).

Primary Key

- Example:

```
CREATE TABLE Movie (
        title CHAR(20),
        year INT,
        length INT,
        PRIMARY KEY(title, year)
);
```

Here, the key declaration must appear on a separate line, since the key consists of more than 1 attribute (title, year)

Foreign Keys

- Referential integrity constraints:

  Intuitively, values for certain attributes must “make sense”

  - That is, every non-Null value in attribute **A** of relation **R** must appear in attribute **B** of relation **S -- Inclusion dependency**
- In SQL, we declare an attribute (or a set K of attributes) in a relation (**R**) to be a *foreign key*, if they reference/point to some attribute(s) G of some related relation **S**.

  That is, t[K] = s[G] if tuple *t* in **R** refers to tuple *s* in **S**

  - The set of attributes G must be declared as the *primary key* of **S**
  - The notion of referential integrity constraint is the one that connects or relates tuples in different relations.

Foreign Keys

- Two ways to declare foreign keys:
  - If the foreign key is a single attribute A, we may write the following after the attribute *name* and its *type*:

(1) **REFERENCES** < referenced-table > (A)

  - When the foreign key includes more than one attribute, write it as a separate line in the **CREATE TABLE** declaration:

(2) **FOREIGN KEY** (<attributes>) **REFERENCES** <table> (<attributes>)

    - Use form (2) if the foreign key includes 2 or more attributes

Foreign Keys

- Relation Schemas:

  Exec(eName, address, cert#, netWorth)

  Studio (sName, address, ↖ presidentC#)

  Here, Studio(presidentC#) refers to Exec(cert#).

```
CREATE TABLE Studio(
        sName CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presidentC#  INT REFERENCES Exec(cert#)
 );
```

Foreign Keys

- Relation Schemas:

  Exec(eName, address, cert#, netWorth)

  Studio (sName, address,presidentC#)

  CREATE TABLE Studio(
  sName **CHAR**(30) **PRIMARY KEY**,
  address **VARCHAR**(255),
  presidentC# **INT**,
  FOREIGN KEY presidentC# **REFERENCES** Exec(cert#));

  In either declaration form, when a value ***v*** appears for presidentC# in a Studio tuple, **v** MUST already exist for an Exec tuple.

  Note: *An exception to this requirement is when **v** is NULL.*

Maintaining Referential Integrity

Exec(eName, address, cert#, netWorth)
Studio (sName, address,presidentC#)

- How to maintain referential integrity when the database is modified?
- Possible situations **violating** foreign key constraints:
  - Insert:
    - 1. Insert a new Studio tuple with presidentC# that is not cert# of any Exec tuple
  - Update:
    - 2. Update a Studio tuple to change its presidentC# to a non-Null value that is not the cert# of any tuple in table Exec
    - 3. Update an Exec tuple **e** that changes **e**.cert#, when the old cert# was presidentC# of some Studio tuple **s** (i.e., **e**.cert# is referenced by **s**)
  - Delete:
    - 4. Delete an Exec tuple when its cert# is presidentC# of 1 or more Studio tuples Recall that cert# is the key of the Exec table

Maintaining Referential Integrity

- There are *three* policies, when there is a transaction that violates a referential integrity:
  - The ***reject*** policy *(default)*

    The system will reject any transaction violating referential integrity constraints -- a run-time error will be generated and the database state will *not* change.
  - If we update (3) or delete (4) a referenced item:
    - The ***cascade*** policy: changes to the referenced attributes are "mimicked" at the foreign key (e.g. presidentC#)
    - The "***set null***" policy: set the referencing attribute to NULL (e.g., presidentC# in Studio)

Maintaining Referential Integrity

Exec(eName, address, cert#, netWorth)

Studio (sName, address,presidentC#)

Example: For the (default) reject policy, the system will **reject** the following modifications/transactions:

- Insert a new Studio tuple with presidentC# that is not in any Exec tuple
- Update a Studio tuple to change the presidentC# to a non-null value that is not in any Exec tuple
- Update an Exec tuple to change cert#, when the old cert# value is presidentC# in some Studio tuple(s)
- Delete an Exec tuple when its cert# is presidentC# in some Studio tuple(s)

Maintaining Referential Integrity

Exec(eName, address, cert#, netWorth)

Studio (sName, address,presidentC#)

- The cascade policy:
  - In case of **deleting** (**updating)** a tuple in a referenced relation (Exec), the DBMS will delete (update) the referencing tuple(s) in Studio
- Example:
  - The system will cascade the following modifications:
    - Update Exec tuple to change cert#, when its cert# is presidentC# in some Studio tuple
      - The system will update presidentC# in corresponding Studio tuple(s)
    - Delete an Exec tuple when its cert# is presidentC# in some Studio tuple
      - The system will delete corresponding tuple(s) in Studio

Maintaining Referential Integrity

- The set-null policy:
  - In case of **deleting** or **updating** a tuple in a referenced relation (Exec), the DBMS will set to NULL the corresponding values in the referencing tuples (in Studio)
- Example:
  - The system will do the following modifications:
    - Update Exec tuple to change cert#, when its cert# is presidentC# in some Studio tuple
      - The DBMS will set to NULL presidentC# in corresponding Studio tuple(s)
    - Delete an Exec tuple when its cert# is presidentC# in some Studio tuple(s)
      - The DBMS will set to NULL presidentC# in the corresponding Studio tuple(s)

Selecting a Policy

- Reaction options/policies may be chosen (by ?) for deletes and updates, in an *independent* way
- **ON [DELETE | UPDATE] [CASCADE | SET NULL]**

Selecting a Policy

- Example:

```
CREATE TABLE Studio(
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presidentC#  INT,
        FOREIGN KEY presidentC# REFERENCES Exec(cert#)
              ON DELETE SET NULL   (or SET DEFAULT)
              ON UPDATE CASCADE
);
```

Selecting a Policy

- "Correct“ or “right” policy is a design decision
- Example:
  - If a studio president retires (and its tuple gets deleted), the studio should exist with a NULL value for the president and not be deleted along with the president
  - If the certificate number cert# of a studio president was changed (by an update), it should be updated in all corresponding places (presidentC# in Studio, in our case); *we should not lose the information on who is the current president of a given studio*

Not-Null Constraints

- We can assert that the value of an attribute may not be NULL
- Example:

  presidentC# **INT REFERENCES** Exec(cert#) **NOT NULL**

Two consequences:

1. We can't insert a tuple into Studio by just specifying name and address; *the value presidentC# must also be present.*

2. We can't use the "set-null" policy to fix foreign key violations by changing presidentC# to be NULL

Attribute-Based Checks

- Aside from “referencing”, more complex constraints can be attached to an attribute declaration, followed by

  CHECK (condition-on-attribute)

- That is, we want the condition to hold for that attribute in every tuple in the relation
- Condition may involve the checked attribute
- Other attributes and relations may be involved, but only in subqueries (see slide #46)
  - **Oracle** : No subqueries allowed in the **condition**

Attribute-Based Checks

- Example:

```
CREATE TABLE Studio(
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presidentC#  INT
       CHECK (presidentC# >= 100000)
 );
```

This requires certificate numbers to be at least 6 digits.

Attribute-Based Checks

When an attribute-based check/condition is (not) done?

- The condition is checked
  - When the associated attribute changes, i.e., an insert or update occurs
- The condition is **NOT** checked
  - When the relations involved in the subquery of the condition are changed; see the next slide

Attribute-Based Checks

- Example:

```
CREATE TABLE Studio(
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        presidentC#  INT
        CHECK (presidentC# IN (SELECT cert# FROM Exec))
 );
```

  - Is this check the same as a foreign-key constraint?
    - Not really! The above check is done only when we insert a tuple in **Studio** or change the presidentC# in an existing tuple in **Studio**, **not** when deleting or update a tuple in **Exec**

Attribute-Based Checks

- Example:

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

The above condition uses an explicit set/relation with two tuples, providing possible values for attribute *gender*

Tuple-Based Checks

- We restrict some components of the tuples of a relation by a **tuple-based check**
- Tuple-based check must appear on a separate element/part in a table declaration command
- Format:

  CHECK (condition)

  - Condition may involve any attribute(s) of the table
  - Other attributes and tables may be involved, but only in subqueries
    - **Oracle**: Does not support subqueries in condition

Tuple-Based Checks

- Example:

```
CREATE TABLE Star(
        name CHAR(30) PRIMARY KEY,
        address VARCHAR(255),
        gender CHAR(1),
        birthdate DATE,
        CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

This constraint says that if a "star is male" (M), then "his name must not begin with Ms." (~L), i,e, M→~L, or ~M ∨ ~L

Modification of Constraints

- We can add, modify, or delete constraints at any time
- Give names to constraints so we can "refer" to them later

Examples of *defining* constraints:

name **CHAR**(30) **CONSTRAINT** Name-Is-Key **PRIMARY KEY**,
OR
CONSTRAINT RightTitle **CHECK** (gender = 'F' **OR** name **NOT LIKE** 'Ms.%')

Examples of *modifying* (deleting/adding) constraints:

ALTER TABLE Star **DROP CONSTRAINT** Name-Is-Key;
OR
ALTER TABLE Star **ADD CONSTRAINT** NameIsKey **PRIMARY KEY** (*name*);

Assertions

- *Assertions*, also called "*general constraints*", are boolean-valued SQL expressions that must *always* hold true
- Sometimes we need a constraint that involves relation as a whole or part of the database schema

  (This is not supported in Oracle)
- Assertions are checked when a specified relation changes
- Syntax:

  CREATE ASSERTION < assertion-name > **CHECK** (< condition >);

  Note: Unlike attribute/tuple based checks, assertions are defined outside table declarations.

Assertions

- Relation schemas:
  Studio (name, address, presidentC#)
  Exec (name, address, cert#, netWorth)
- Assertion in SQL:

```
CREATE ASSERTION RichPresident CHECK
      (NOT EXIST (SELECT *
                  FROM Studio, Exec
                  WHERE Studio.presidentC# = Exec.cert# AND
                        Exec.netWorth < 10000000
                  )
      );
```

  - This constraint is checked when **Studio** and/or **Exec** tables change

Triggers

- Also called **event-condition-action (ECA)** rules
- *Event*
  - DB transactions (Insertion, Deletion, Update)
- *Condition*
  - A condition to check if a trigger applies
- *Action*
  - One or more SQL statements to be executed if a triggering event occurred and the condition(s) holds.

Triggers

- Triggers are not supported in SQL2
- Oracle's version and SQL3 version differ from Checks or SQL2 assertions in that:
  - For triggers, event is "programmable", rather than **implied** by the kind of check
  - For checks, we don't have conditions specified
  - (Re)action could be any sequence of database operations

➔Active Database Management Systems (ADBMS)

Triggers

Relation Schema: **Exec(name, address, cert#, networth)**

Example of an SQL trigger:

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec                    ← Event
REFERENCING
    OLD ROW AS OldTuple,
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)        ← [Condition]
    UPDATE Exec                                     ← Action
    SET netWorth = OldTuple.netWorth
    WHERE cert# = NewTuple.cert#;
```

Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- AFTER could be changed to either:
  - BEFORE
  - INSTEAD OF

Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- **OF** in **'UPDATE OF'** is optional
  - If present, it defines the event to be only an update of the attribute(s) listed after the keyword **OF**

Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
    OLD ROW AS OldTuple,
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
    UPDATE Exec
    SET netWorth = OldTuple.netWorth
    WHERE cert# = NewTuple.cert#;
```

- UPDATE (OF whatever) could be changed to:
  - INSERT
  - DELETE

Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER INSERT ON Exec
REFERENCING
    OLD ROW AS OldTuple,
    NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
    UPDATE Exec
    SET netWorth = OldTuple.netWorth
    WHERE cert# = NewTuple.cert#;
```

← **OLD makes no sense!**

- If the event is **INSERT**:
  - **REFERENCING**
    NEW ROW AS NewTuple

Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER DELETE ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth )
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

← NEW makes no sense!

- If event is **DELETE**:
  - **REFERENCING**
    OLD ROW AS OldTuple

Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
 WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert# ;
```

- The action could be any sequence of SQL statements, separated by semicolons and embeded in a pair of **BEGIN** and **END**, **e.g.,** **WHEN condition BEGIN S1; … Sk; END;**

Triggers

```
CREATE TRIGGER NetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
   OLD ROW AS OldTuple,
   NEW ROW AS NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
   UPDATE Exec
   SET netWorth = OldTuple.netWorth
   WHERE cert# = NewTuple.cert#;
```

- This trigger is **row-level trigger**
- If we omit "**FOR EACH ROW**", the trigger becomes a **statement-level trigger,** which is the default

Triggers

- If we update an “entire” table with an SQL statement
  - A **row-level trigger** will be executed once for each tuple
  - A **statement-level trigger** will be executed only once for the entire update
- In a statement-level trigger:
  - We can not refer to old and new tuples
  - Instead, we can/should refer to
    - The collection of old tuples as **OLD TABLE**
    - The collection of new tuples as **NEW TABLE**

Triggers

- **Ex: Constraint: "The *average* net worth of executives may NOT drop below $500,000." This could be violated by updating netWorth, or deleting or inserting tuples from/into Exec.**
- **Below is a trigger for update; similar triggers must be written for delete and insert.**

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF netWorth ON Exec
REFERENCING
    OLD TABLE AS OldStuff,
    NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN (500000 > (SELECT AVG(networth) FROM Exec))
BEGIN
   DELETE FROM Exec
   WHERE (name, address, cert#, netWorth) IN NewStuff ;
   INSERT INTO Exec(SELECT * FROM OldStuff) ;
END;
```

Triggers

Relation scheme: Employee(name, empId, salary, dept, supervisorId)

Constraint: *No employee gets a salary more than his/her supervisor.*

```
CREATE TRIGGER Inform_supervisor
BEFORE INSERT OR UPDATE OF salary, supervisorId ON Employee
NEW ROW AS new
FOR EACH ROW
WHEN (new.salary > (SELECT salary
                    FROM Employee
                    WHERE empId=new.supervisorId))
Begin
    ROLLBACK;
    Inform_Supervisor(new.supervisorId, new.empId);
End;
```

Instead-Of Triggers

Relation Scheme: Movie(title, year, length, filmType, studioName, producerC#)

```
CREATE VIEW ParamountMovie AS
SELECT title, year
FROM Movie
WHERE studioName = 'Paramount';
```

The following *trigger* replaces an insertion on the view (ParamountMovie) with an insertion on its underlying base table (Movie)

```
CREATE TRIGGER ParamountInsert
INSTEAD OF INSERT ON ParamountMovie
REFERENCING NEW ROW AS NewRow
FOR EACH ROW
INSERT INTO Movie(title, year, studioName)
VALUES (NewRow.title, NewRow.year, 'Paramount');
```

A test on Triggers!

Suppose relation **rel(a, b)** has no tuples. Consider the trigger:

```
CREATE TRIGGER T
    AFTER INSERT ON rel
    REFERENCING NEW ROW AS newRow
    FOR EACH ROW
    WHEN (newRow.a * newRow.b > 10)
    INSERT INTO rel VALUES(newRow.a - 1, newRow.b + 1);
```

Inserting which of the following tuples into relation **rel** results in having exactly 2 tuples in **rel**?

1. (3, 5)
2. (4, 3)
3. (3, 4)
4. (5, 3)