HA NOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

ĐẠI HỌC BÁCH KHOA HÀ NỘI
VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

IT3090E - Databases

Chapter 7: Dynamic constraints – Rules & triggers

Muriel VISANI
murielv@soict.hust.edu.vn

LEARNING POINTS

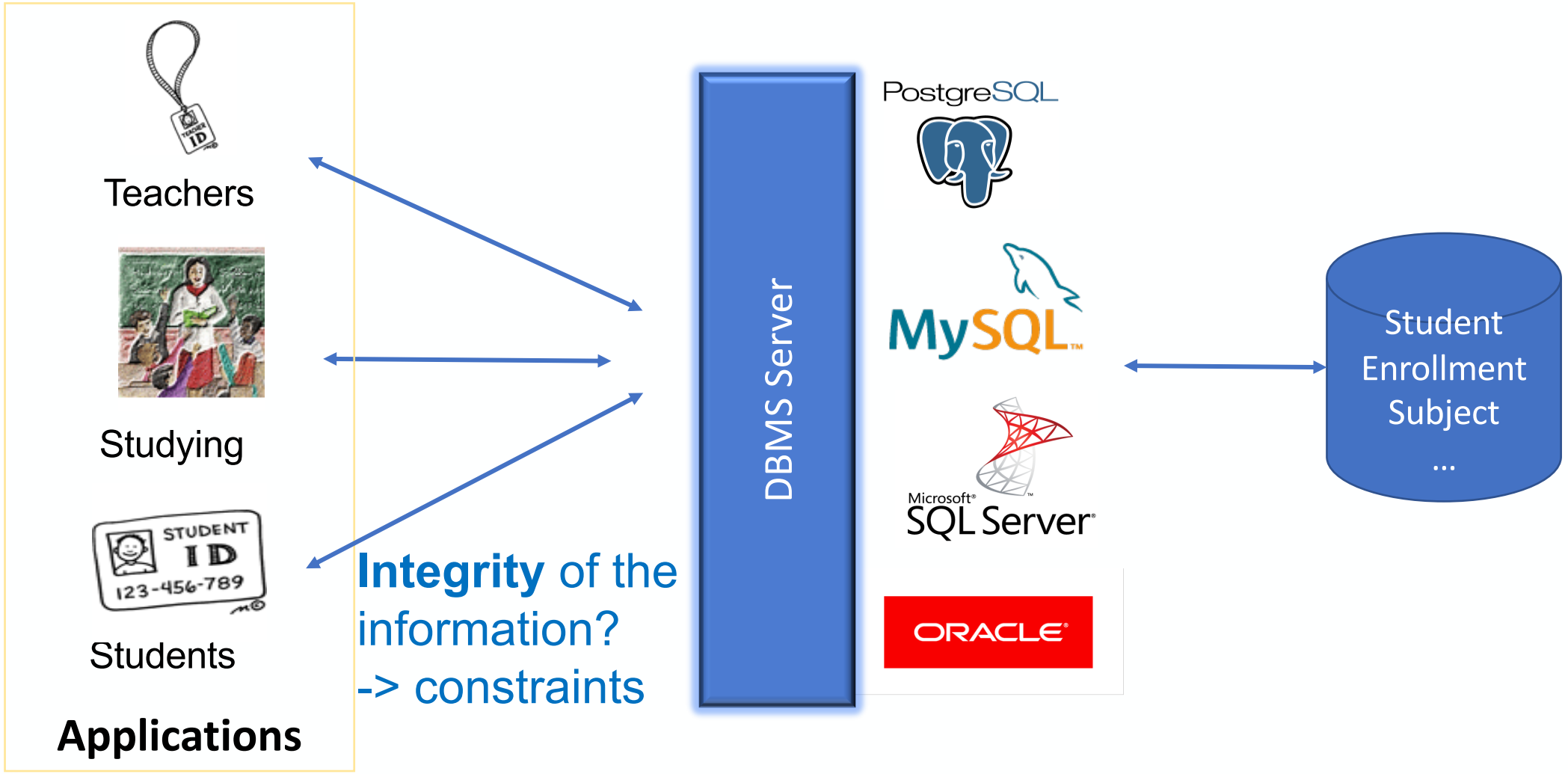
1. Static *vs.* dynamic constraints
2. Rules
3. Triggers

LEARNING OBJECTIVES

Upon completion of this lesson, students will be able to:

1. Know about different constraints and define them correctly
2. Understand rules and triggers:
   - What is a rule/trigger? how does it works ? When using them?
3. Define simple triggers

1. Static *vs.* dynamic constraints



1. Static *vs.* dynamic constraints

- ***Database Schema used in this lecture***

student(**student_id**, first_name,last_name, dob, gender,

address, comment, email, *clazz_id#*)

clazz(**clazz_id**, name, *lecturer_id#*, *monitor_id#*, number_students)

subject(**subject_id**, name, credit, percentage_final_exam)

enrollment(***student_id#, subject_id#*, semester**, midterm_score, final_score)

lecturer(**lecturer_id**, first_name, last_name, dob, gender, address, email)

teaching(***subject_id#, lecturer_id#***)

grade(**code**, from_score, to_score)

1. Static *vs.* dynamic constraints

- A static constraint is a constraint on data elements that the DBMS is required to enforce, whatever happens in the database
  - Example: key constraints, CHECK constraints
- A dynamic constraint is a constraint **between different attributes** that is triggered by some triggering event
  - The concerned attributes might come from a single table, or from different tables
  - Triggering events apply to tables (insert, update, delete…)
  - To enforce dynamic constraints, one might use rules, or triggers

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints***
  - Keys: PRIMARY KEY vs. UNIQUE
  - Foreign keys – referential integrity
  - Attribute-based CHECK
    - Constraints on the values of a particular attribute.
  - Tuple-based CHECK (multi-attribute based checks)
    - Constraints between the values of different attributes.
  - Assertions
    - Constraint verifying any SQL Boolean expression

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : PK* vs UNIQUE**
  - Declaring (similar syntax):

```
CREATE TABLE student (
    student_id CHAR(8) NOT NULL,
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20) NOT NULL,
    ...
    email varchar(50) UNIQUE,
    clazz_id CHAR(8),
    CONSTRAINT student_pk PRIMARY KEY (student_id));
```

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : PK* vs UNIQUE**
  - Declaring (similar syntax):

```
CREATE TABLE student (
    student_id CHAR(8) NOT NULL PRIMARY KEY,
    first_name VARCHAR(20) NOT NULL,
    last_name VARCHAR(20) NOT NULL,
    ...
    email varchar(50),
    clazz_id CHAR(8),
    CONSTRAINT student_uk UNIQUE(student_id));
```

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : PK* vs UNIQUE**

	PRIMARY KEY	UNIQUE KEY
Max number / table	One	Multiple
NULL columns allowed	No	Yes
Default index	CLUSTERED	NON-CLUSTERED
Purpose	Enforce Entity Integrity	Enforce Unique Data
Number of columns	One or more columns	One or more columns
Can be referenced by a Foreign Key Constraint	Yes	Yes

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : FOREIGN KEYS***
  - Referenced attributes must be declared as PRIMARY KEY or UNIQUE
  - The referencing and referenced attribute(s) must have the same type:

```
CREATE TABLE clazz (
   clazz_id CHAR(8) NOT NULL PRIMARY KEY,
    name VARCHAR(20), … );

CREATE TABLE student (
    student_id CHAR(8) NOT NULL,
    ... ,
    clazz_id CHAR(8),
    CONSTRAINT student_pk PRIMARY KEY (student_id));

ALTER TABLE student ADD CONSTRAINT student_fk_class
FOREIGN KEY (clazz_id) REFERENCES clazz(clazz_id);
```

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : FOREIGN KEYS***
  - ***Enforcing FK constraints***
    - An insert or update to student that introduces a non-existent clazz_id (clazz_id value is not found in clazz)

      ➔ **Raises an error (the insert or update is prevented)**
    - A deletion or update to clazz that removes a clazz_id value found in some tuples of student:
      - By default (RESTRICT): reject the modification
      - CASCADE: make the same changes in student
      - SET NULL: change clazz_id in student to NULL
      - SET DEFAULT: change clazz_id in student to its default value

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : FOREIGN KEYS***
  - ***Choosing a policy – might differ on update, on delete***
    - Example:

```
ALTER TABLE student
   ADD CONSTRAINT student_fk_class FOREIGN KEY
(clazz_id) REFERENCES clazz(clazz_id)
   ON DELETE SET NULL
   ON UPDATE CASCADE;
```

1. Static *vs.* dynamic constraints – static constraints

● *Recalls/ complements about static constraints : checks constraints*

- A check constraint is a type of integrity constraint in SQL which specifies a requirement that must be met by each row in a database table. The constraint must be a predicate. It can refer either:
  - to a single column **of the table**: attribute-based checks
  - to multiple columns **of the table**: tuple-based checks

- Timing of checks: on insert or update only.

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : Attribute-based checks***

  - Constraints on the value of a particular attribute
    - Add CHECK(<condition>) to the declaration for the attribute or declare as a constraint (with name)
    - Many DBMS **DO NOT SUPPORT** sub-queries in CHECK constraints!
  - Example:

```
CREATE TABLE student (
    student_id CHAR(8) NOT NULL PRIMARY KEY, ...,
    gender CHAR(1),
    clazz_id CHAR(8) CHECK (clazz_id LIKE 'P__-____'),
    CONSTRAINT student_chk_gender CHECK (gender = 'F' OR gender = 'M'));
```

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : tuple-based checks***

  - CHECK (<condition>) may be added as a **relation-schema element.**
  - The condition may refer to any attribute of the relation

```
CREATE TABLE grade(
    code CHAR(1) NOT NULL,
    from_score DECIMAL(3,1) NOT NULL,
    to_score DECIMAL(3,1) NOT NULL, ...,
    CONSTRAINT grade_chk_toScore CHECK (to_score >
from_score) );
```

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : checks constraints***
  - ***Timing of checks***
    - For any type of check constraints (attribute-based or tuple-based)…
      - The check operation is performed only when a value **for the attribute of the check** is inserted or updated
      - A (silly) example:

```
CREATE TABLE clazz (
        clazz_id CHAR(2) NOT NULL PRIMARY KEY,
        name VARCHAR(20));

CREATE TABLE student (
        student_id CHAR(2) NOT NULL,
        clazz_id CHAR(2) CHECK (clazz_id >student_id),
        CONSTRAINT student_pk PRIMARY KEY (student_id));
```

```
ALTER TABLE student ADD CONSTRAINT student_fk_class
FOREIGN KEY (clazz_id) REFERENCES clazz(clazz_id) ON
UPDATE CASCADE;

insert into clazz VALUES('11');
insert into student VALUES('01', '11');

update student set student_id='22' where student_id='01';
```

➔ **Raises an error (the update is prevented)**

```
update student set clazz_id='00' where clazz_id='11';
```

➔ **Raises an error (the update is prevented)**

```
update clazz set clazz_id='00' where clazz_id='11';
```

➔ **Raises an error (updates in clazz and student prevented)**

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : ASSERTIONS***
  - ***Declaring***
    - Database-schema elements, like relations or views
      - Can implement sub-queries in CHECK constraints (from other tables also)
    - Many DBMS **DO NOT SUPPORT ASSERTIONS!!!**
    - (When supported), they can be defined by:

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

      - <condition> may refer to **any relation or attribute** in the database schema
    - Drop an assertion:

```
DROP ASSERTION <assertion name>;
```

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : ASSERTIONS***
  - ***Examples***

```
CREATE ASSERTION teachingSubject CHECK (
   (SELECT COUNT(*) FROM teaching) >=
   (SELECT COUNT(*) FROM subject) );

CREATE ASSERTION numberStdInClass CHECK (
   NOT EXISTS (
      SELECT * FROM clazz c
      WHERE number_students <>
         (SELECT count(*) FROM student
            WHERE clazz_id = c.clazz_id) ) );
```

1. Static *vs.* dynamic constraints – static constraints

- ***Recalls/ complements about static constraints : ASSERTIONS***
  - ***Timing of Assertion Checks***
    - In principle, we must check every assertion after every modification to any relation of the database
    - A clever system should observe that only certain changes could cause a given assertion to be violated
      - No change to **student** can affect **teaching Subject**
      - Neither can an insertion to **teaching**
    - Many DBMS **DO NOT SUPPORT ASSERTIONS!!!**
      - **They are difficult to implement effectively in practice (especially their timing)**

1. Static *vs.* dynamic constraints – dynamic constraints

- A dynamic constraint is a constraint **between different attributes** that is triggered by some triggering event
  - The concerned attributes might come from a single table, or from different tables
  - Triggering events apply to tables (insert, update, delete…)
  - To enforce dynamic constraints, one might use rules, or triggers
- There are two main DB structures to enforce dynamic constraints:
  - Rules
    - For simple processing
  - Triggers
    - For more complex processing operations

1. Static *vs.* dynamic constraints – dynamic constraints

- Example:
  - We have two tables: bank accounts and their operations (payments, receiving money)
  - We need to implement one dynamic constraint to ensure this dataset's integrity
    - Which one?
    - Solution:

```
create table account(account_id integer, balance decimal check (balance >1000));
create table operations(account_id integer,  amount decimal);

insert into account values (123,1100);
insert into operations values(123, -10);
```

1. Static *vs.* dynamic constraints – dynamic constraints

- **NEW** *vs.* **OLD**
  - **NEW** and **OLD** are used when implementing dynamic constraints
    - They depend on the type of triggering event
    - They refer to the table concerned by the triggering event
      - They can be used to replace the name of this table by its old (or new) version, before (or after) the event
  - When processing a dynamic constraint, we need to distinguish between
    - The new values (during an INSERT or UPDATE): **NEW**
    - The old values (during an UPDATE or a DELETE): **OLD**

2. Rules

- **INSTEAD** *vs.* **ASLO**
  - Two keywords used in rules to say whether the processing will be applied **on top of** the triggering event, or **instead of** the triggering event
  - The **processing** might be NOTHING, or any DML operation (insert, update, delete…)

- Declaring a rule:

```
CREATE RULE <name_rule> AS ON <triggering_event>  TO <table_name>
DO  ALSO | INSTEAD <processing>
```

2. Rules

- Example 1: forbidding a given user some DML operation

```
CREATE RULE refuse_insert_shop
AS ON  INSERT
TO shop
WHERE current_user <> 'Visani'
DO INSTEAD NOTHING;
```

- Triggering the rule (by user 'Visani' only):

```
INSERT INTO shop values('SH11111S','17000');
--INSERT 0 0 Query returned successfully in 107 msec.
```

2. Rules

- Example 2: we want to keep a history of all the replacements of 10+ parts

```
create table history (reference text, interv_no integer, qtity smallint)

CREATE RULE archiving_big_remplacements
AS ON  DELETE
TO REPLACEMENTS
WHERE OLD.qteremplacee > 10
DO ALSO (insert into history values (OLD.reference, OLD.interv_no, OLD.qtity) );
```

2. Rules

•Exercise: write the rule needed to update the account balance based on the operations on that account, from the two following tables:

```
create table account(account_id integer, balance decimal check (balance >1000));

create table operations(account_id integer,  amount decimal);
```

Idea: trigger the rule, then the check, to avoid the balance to go below 1000 (for now, we just focus on the integrity of the table balance, not on operations), as follows:

```
-- Triggering (implicitely) the rule, then the check:
insert into account values (123,1100);
insert into operations values(123, -200);
--ERROR: new row for relation "account" violates check constraint
"account_balance_check" DETAIL: Failing row contains (123, 900). SQL state: 23514
```

2. Rules

- Solution:

2. Rules

- Problem: if the operation is negative (withdraw/payment) and exceeds the limit, then it does not change the balance but the operation itself is still inserted in the database
- Exercise:
  1- create another rule to forbid such operations to be inserted in the database
  2- Modify the first rule so that it's only triggered when necessary
    - For better efficiency
    - To avoid the error being raised due to the check constraint on the table account

2. Rules

- Solution: 1- create another rule

-- Triggering the rule, then the check:
insert into account values (123,1100);
insert into operations values(123, -200);

-- INSERT 0 0 Query returned successfully in 70 msec

-> The table operations is not modified

2. Rules

- Solution: 2- modify the old rule

2. Rules

- Problem: with this solution, we totally loose track of the operations that were rejected
- Question:
  - What do you propose to solve this problem?
- Solution:

2. Rules

- Another problem: when an operation is rejected, we cannot raise a notice to show a warning to the user
  - That is one of the limitations of rules: we can only use SQL DML in the ALSO / INSTEAD

- Other limitations of rules include:
  - Rules are widely used with many DBMS, but SQL Server warns that it's not going to be included in the future versions
    - But this is since 2012 and it is still in the version 2019 ;-)

3. Triggers

3.1. What is a database trigger?

3.2. Trigger Definition

3.3. Using triggers

3.4. Examples

3. Triggers

3.1. What is a database trigger?

- **Motivation**
  - **Attribute- and tuple-based checks**
    - checked at known times (insert, update),
    - but are not powerful (in particular, most DBMS don't authorize nested queries)
  - **Assertions**
    - In principle, powerful,
    - but many DBMS don't support them
    - And, the DBMS often can't tell when they need to be checked
  - **Triggers** let the **user decide** when to check for any condition (even complex)

3. Triggers

3.1. What is a database trigger?

- Definition: a trigger is
  - A procedure (stored or not) associated with a table
  - Is executed automatically every time the triggering event occurs
    - Either **BEFORE** the trigger event is realized
      - Often to prevent or modify it
        - *With rules, if we wanted to implement BEFORE, we had to use INSTEAD*
    - Or **AFTER** the realization of the trigger event
      - Often to show a warning or trigger other events
        - *With rules, if we wanted to implement AFTER, we had to use ALSO*

3. Triggers

3.1. What is a database trigger?

ECA Rules

- A trigger defines an operation that is performed when a specific event occurs on a relation:
  - inserts a new record / updates an existing record / deletes a record
- Trigger functions have access to special variables from the database engine
- Called also ECA rules **(Event-Condition-Action)**
  - Event: type of database modification
  - Condition: Any SQL Boolean-valued expression
  - Action: Any SQL statements (not only DML statement)

3. Triggers

3.1. What is a database trigger?

- **Example (the syntax differ depending on the DBMS)**

  - **Constraint:** when a new student is inserted into student relation, the number of students in his class must be increased

    student(**student_id**, first_name,last_name, dob, gender, address, comment, email, *clazz_id#*)

    clazz(**clazz_id**, name, *lecturer_id#*, *monitor_id#*, number_students)

```
CREATE TRIGGER clazz_changes_tg
AFTER INSERT ON student
REFERENCING NEW ROW AS nnn
FOR EACH ROW
WHEN (nnn.clazz_id IS NOT NULL)
BEGIN
    update clazz
    set number_students = number_students + 1
    where clazz_id = nnn.clazz_id;
END;
```

Event

Condition

Action

NNN replaces NEW

3. Triggers

3.2. Trigger Definition

Syntax

- Creating a trigger:

```
CREATE [OR REPLACE] TRIGGER <trigger_name>
    {BEFORE | AFTER | INSTEAD OF }
    {INSERT | DELETE | UPDATE [OF <attribute_name>]}
    ON <table_name>
    REFERENCING {NEW | OLD} {ROW | TABLE} AS <name>
    [FOR EACH ROW | STATEMENT]
    [WHEN (<condition>) ]
    BEGIN
        <trigger body goes here >
    END;
```

- Dropping a trigger:

```
DROP TRIGGER <trigger_name>;
```

3. Triggers

3.2. Trigger Definition

Trigger event and temporality

- **INSERT, DELETE, UPDATE , UPDATE OF**
  - UPDATE OF <colums>: update on a particular attribute
- **AFTER, BEFORE, INSTEAD OF** (the triggering event)**:**
  - AFTER, BEFORE: used for tables / views
  - INSTEAD OF: used only for views
    - A way to execute view modifications: triggers translate them to appropriate modifications on the base tables

3. Triggers

3.2. Trigger Definition

Triggers level

- Row-level trigger:
  - Indicated by option FOR EACH ROW
  - Trigger is executed once for each modified tuple
- Statement-level trigger:
  - Without option FOR EACH ROW (default), or with FOR EACH STATEMENT
  - Trigger is executed only once for each SQL statement, regardless of how many tuples are modified

3. Triggers

3.2. Trigger Definition

● REFERENCING

- **INSERT** statements imply a new tuple (for row-level) or new table (for statement-level)
  - The table is the set of inserted tuples
- **DELETE** implies an old tuple or table
- **UPDATE** implies both
- If possible, you can refer to the OLD and NEW values by :

  REFERENCING [NEW | OLD] [TUPLE | TABLE] AS <name>

- Each DBMS has its own implementation, REFERENCING may not be used:
  - Access directly to special variables from the database engine: NEW, OLD,..

3. Triggers

3.2. Trigger Definition

Condition

- Any boolean-valued condition
- Evaluated on the state of the database as it would exist before or after the triggering event, depending on whether BEFORE or AFTER is used.
  - But always before the changes take effect.
- Access the new/old tuple/table through the names in the REFERENCING clause

3. Triggers

3.2. Trigger Definition

Action

- Can be more than one SQL statement:
  - Surrounded by BEGIN .. END
- Language:
  - Either, simple SQL statements
  - Or, extension of SQL: procedural languages, depends on each DBMS
    - PL/SQL (Oracle), PL/pgSQL (PostgreSQL), T-SQL(SQL Server) ,..

3. Triggers

3.3. Using triggers

When?

- Auditing data modification (keeping history of data), providing transparent event logging
- Validation and business security checking
  - Eg. column formatting before and after inserts into database
- Enforcing complex, dynamic integrity constraints
- Enforcing complex business rules
- Maintaining replicate tables (not advised in most cases)
- Building complex views that are updatable
- …

3. Triggers

3.3. Using triggers

● Guidelines for designing triggers

- Do not define triggers that duplicate database features
  - do not define triggers to reject bad data if you can do the same checking through constraints
- Use triggers only for centralized, global operations that must be fired every time the triggering event occurs, regardless of which user or database application issues the event
- Do not create recursive triggers
- Use triggers on DATABASE in the right cases only (server error, logon, logoff,...):
  - they are executed for every user, every time the triggering event occurs

3. Triggers

3.4. Examples with different DBMS

Oracle

- Add a new column in **clazz** relation

```
alter table clazz
add column number_students integer not null default 0;
```

- Create a trigger on **student** relation

```
CREATE TRIGGER clazz_changes_tg
AFTER UPDATE ON student
FOR EACH ROW
WHEN (:NEW.clazz_id <> :OLD.clazz_id)
BEGIN
    update clazz set number_students= number_students+1
    where clazz_id = :NEW.clazz_id;
    update clazz set number_students = number_students-1
    where clazz_id = :OLD.clazz_id;
END;
```

3. Triggers

3.4. Examples with different DBMS

- **PostgreSQL**

```
CREATE FUNCTION public.tg_fnc_change_clazz()
    RETURNS trigger LANGUAGE 'plpgsql' AS $$
BEGIN
   update clazz set number_students = number_students+1
   where clazz_id = NEW.clazz_id;
   update clazz set number_students = number_students-1
   where clazz_id = OLD.clazz_id;
   return NEW;
END; $$

CREATE TRIGGER tg_af_update_clazz
    AFTER UPDATE OF clazz_id
    ON student
    FOR EACH ROW
    EXECUTE PROCEDURE public.tg_fnc_change_clazz();
```

Summary

- Constraints, Assertions, Rules, Triggers:
  - How to declare
  - Timing of checks
  - Differences
- Only use them if you really need to, especially triggers
- Each DBMS has its own variation in implementation:
  - Options
  - Syntax: triggers as an example

  ➔ Reading documentation for each DBMS installed

References

- Hector Garcia-Molina, Jeffrey D. Ullman, Jennifer Widom. Database Systems: The Complete Book. Pearson Prentice Hall. the 2nd edition. 2008: Chapter 7
- Nguyen Kim Anh, Nguyên lý các hệ cơ sở dữ liệu, NXB Giáo dục. 2004: Chương 7

ĐẠI HỌC BÁCH KHOA

25 YEARS ANNIVERSARY SOICT

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

Thank you for your attention!

soict.hust.edu.vn/ fb.com/groups/soict