COMP9120

Week 5: Database Integrity

Semester 1, 2025

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents co-developed with Dr Lijun Chang

Warming up

THE UNIVERSITY OF SYDNEY

Acknowledgement of Country

I would like to acknowledge the Traditional Owners of Australia and recognise their continuing connection to land, water and culture. I am currently on the land of the Gadigal people of the Eora nation and pay my respects to their Elders, past, present and emerging.

Copyright warning

Outline

- **Overview of Integrity Constraints**

- **Static Integrity Constraints**
  - Domain Constraints
  - Key / Referential Constraints
  - Semantic Integrity Constraints
  - Assertions

- **Dynamic Integrity Constraints**
  - Triggers

Integrity Constraints (ICs)

› **Integrity Constraint (IC)**:

A **condition** that must hold **true** for **every instance** of a database:

- ICs are an *integral* part of the initial database *schema design* (through the **create table** command) to ensure the database **integrity** and **consistency**.
- ICs can be *added/updated* at any time (through **alter table** *table-name* **add/alter** *constraint* command).
  - When such a command is executed, the system *first ensures* that the relation *satisfies* the specified *constraint*. If it *does*, the constraint is *added* to the relation; if *not*, the command is *rejected*.
- A **legal** instance of a relation is one that satisfies **all** specified **ICs**.
  - Not *necessarily at all times.*

Integrity Constraints (ICs)

- Example of integrity constraints
  - Each student ID must be *unique*.
  - No two lecturers can have the *same* ID.
  - Every school name in the *Unit* relation must have a matching school name in the *School* relation.
  - For every student, a *name* must be given.
  - The *only* possible grades are either ‘F’, ‘P’, ‘C’, ‘D’, or ‘H’.
  - *Valid* lecturer titles are ‘Associate Lecturer’, ‘Lecturer’, ‘Senior Lecturer’, ‘Associate Professor’, or ‘Professor’.
  - Students can *only* enrol in the units of study that are *currently* on offer.
  - The *sum* of all marks in a course *cannot* at *any time* be *higher* than 100.

Capturing ICs

› **Why** do we need to **capture integrity constraints?** To ensure:

1. Data is stored in *accordance* with the *real-world meaning* (semantics) of the domain application by:
   - Avoiding *data entry errors* (e.g., inserting a *grade* into the **Student** table which *does not* exist).
2. *Data consistency* (e.g., *deleting* an employee from the *Employee* table should *also* result in all corresponding tuples from the *Works-on* relation to be *deleted*).
3. *Easier* application *development* and better *maintainability:*
   - Allows for the ICs to be **centrally managed** by the DBMS:
     - We do ***not*** have to ***worry*** about ***how*** integrity constraints are ***enforced/implemented***.

Integrity Constraints in a Database

- As previously indicated, ICs are *specified* as part of the database *schema design*
  - Note: The database designer is still *responsible* for ensuring that the integrity constraints *do not contradict* each other!
    - *The process of detection of IC conflicts* may be *automated,* but this may introduce ***unacceptable computational overhead***.
- Note: ICs are *checked* when the *mentioned parts* of the database are *modified.*
  - We can however specify *when* ICs should be *checked*: e.g., right *after* a SQL statement or at the *end* of a 'transaction'
    - *Transaction*: a **group of statements,** i.e., SQL statements**,** that are executed as *one single undivided* operation, i.e., **atomically**
- What are the possible *ways to react* if an IC is *violated*?
  1. *Reject* the database operation – don't execute it.
  2. *Abort* the whole transaction – *rollback all* operations which are part of the transaction
  3. Execution of *"maintenance" operations* to make DB legal again

An Informal Introduction to Transactions

Definition of a transaction: A *group of statements* that are *executed atomically*

```
BEGIN;
   A group of SQL statements;
COMMIT;
```

- Note: An *SQL statement* starts with predefined keywords and ends with a semicolon`;' . These usually start with the keywords:
  - **SELECT** - extracts data from a database
  - **UPDATE** - updates data in a database
  - **DELETE** - deletes data from a database
  - **INSERT INTO** - inserts new data into a database
  - **CREATE TABLE** - creates a new table
  - **ALTER TABLE** - modifies a table
  - **DROP TABLE** - deletes a table

- Consider an empty table

  R(id: integer, name: varchar(8))

  - What will be the result of the *following transaction*?

    BEGIN;
    INSERT INTO R **VALUES**(1, 'Adam');
    INSERT INTO R **VALUES**(1, 'Smith');
    COMMIT;
  - Can you explain *why*?

Types of Integrity Constraints

› ***Two Types:***

1. ***Static* Integrity Constraints**

   They describe conditions that *every legal instance* of a database *must satisfy*: *Static* integrity constraints are *state* ***independent*****.**

   - Inserts / deletes / updates which *violate* ICs are *disallowed*
   - **Four types** of static integrity constraints:
     - ***Domain Constraints***
     - ***Key Constraints & Referential Integrity***
     - ***Semantic Integrity Constraints***
     - ***Assertions***

2. **Dynamic Integrity Constraints**

   They are predicates on *database state changes* which capture constraints *over two or more states.* Therefore, dynamic integrity constraints are *state* ***dependent***.

   - ***Triggers***

Outline

- Overview of Integrity Constraints

  - **Static Integrity Constraints**
    - Domain Constraints
    - Key / Referential Constraints
    - Semantic Integrity Constraints
    - Assertions

  - Dynamic Integrity Constraints
    - Triggers

Domain Constraints

- **Fields** must be of the *right data domain*
  - *Always* enforced for values that are *inserted* in the database
  - Queries are tested to ensure that the *comparisons* make *sense*.
- Put simply, each attribute needs to have a *data type*

- SQL DDL allows *domains* of attributes to be further *restricted* in the **CREATE TABLE** statement, using the following clauses:
  - **DEFAULT** *default-value*
    default value for an attribute if its value is *omitted* in an insert statement.
  - **NOT NULL**
    attribute is ***not allowed*** to become NULL
  - **NULL**

    the value of an attribute ***may be*** NULL (which is the default)

Example of Domain Constraints

```
CREATE TABLE Student
(
    sid       INTEGER      NOT NULL,
    name      VARCHAR(20)  NOT NULL,
    semester  INTEGER      DEFAULT 1,
    birthday  DATE         NULL,
    country   VARCHAR(20)
);
```

Example:

```
INSERT INTO Student(sid,name) VALUES (123,'Peter');
```

Student				
sid	name	semester	birthday	country
123	Peter	1	null	null

Domain Check Constraints

- *Limit* the *allowed values* for an attribute by specifying *extra conditions* using an *in-line check* constraint

  att-name sql-data-type **CHECK(***condition***)**

- Examples:
  - Grade can *only* be ‘F’,’P’,’C’,’D’, or ’H’

    grade **CHAR**(1) **CHECK(** grade **IN** (‘F’,’P’,’C’,’D’,’H’)**)**
  - Age *must* be positive

    age **INTEGER CHECK(** age >= 0 **)**

User-Defined Domains

› *New domains* can be created from *existing data domains*, with their own defaults and restrictions

CREATE DOMAIN domain-name sql-data-type ...

- Example:

```
CREATE DOMAIN Grade CHAR(1) DEFAULT 'P' CHECK(VALUE IN
                                        ('F','P','C','D','H'))
```

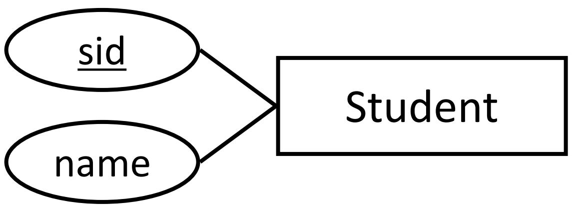
Which is equivalent to:

```
CREATE TABLE Student (
    sid       INTEGER         NOT NULL,
    name      VARCHAR(20)     NOT NULL,
    grade     Grade,
    birthday DATE);
```

```
CREATE TABLE Student (
    sid       INTEGER         NOT NULL,
    name      VARCHAR(20)     NOT NULL,
    grade     CHAR (1) DEFAULT  'P' CHECK (grade IN ('F','P','C','D','H')),
    birthday DATE);
```

Key Constraints

- In SQL, we specify *key* constraints using the **PRIMARY KEY** and **UNIQUE** clauses:

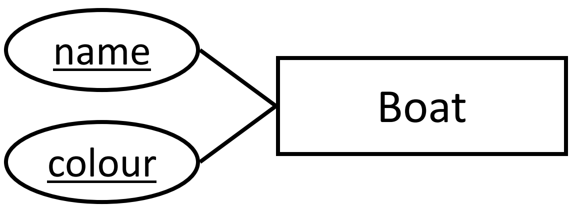



```
CREATE TABLE Student
(
  sid     INTEGER PRIMARY KEY,
  name VARCHAR(20)
);
```

- A **primary key** is automatically **UNIQUE** and **NOT NULL**
  - A relation can have ***multiple*** *candidate* (*unique*) keys, but ***only*** one *primary* key

Key Constraints

- Composite keys: a key consisting of multiple attributes
  - Must be specified in a separate clause




```
CREATE TABLE Boat
(
  name   VARCHAR(20),
  colour  VARCHAR(20),
  PRIMARY KEY (name, colour)
);
```

```
CREATE TABLE Boat
(
  name   VARCHAR(20) PRIMARY KEY,
  colour VARCHAR(20) PRIMARY KEY
);
```

Foreign Keys and Referential Integrity

› **Foreign key:** set of attributes in a *referring* relation that is used to `*refer*' to a tuple in a *parent/referred* relation.

- Must refer to a **candidate key** of the parent (i.e., *referred*) relation

› **Definition of Referential Integrity**: For each tuple in the *referring* relation (called ***child*** or ***dependent table***) whose foreign key value is **a**, there **must** be a tuple in the *referred* relation (called ***parent table***) whose value of the referred attribute is also **a**

e.g. Enrolled(*sid*: integer, ucode: char(8), semester: char(2))
Student(*sid:* integer, name: varchar(20), age: integer, country: char(3))
sid is a foreign key in *Enrolled* referring to the table *Student*:

- If all *foreign key constraints* are *enforced,* then referential integrity is *achieved*, i.e., *no dangling* references

Foreign Keys in SQL

› Express the constraint that *only students listed* in the Student relation should be allowed to *enrol* in Units of study.

```
CREATE TABLE Enrolled
(   sid INTEGER,  uos CHAR(8),  grade VARCHAR(2),
    PRIMARY KEY (sid,uos),
    FOREIGN KEY (sid) REFERENCES Student,
    FOREIGN KEY (uos) REFERENCES Unitofstudy
);
```

Dependent Table

Student

sid	name	age	country
53666	Jones	19	AUS
53650	Smith	21	AUS
54541	Ha Tschi	20	CHN
54672	Loman	20	AUS

Enrolled

sid	uos	grade
53666	COMP5138	CR
53666	INFO4990	CR
53650	COMP5138	P
53666	SOFT4200	D
~~54221~~	~~INFO4990~~	~~F~~

??? Dangling reference

Parent Table

Enforcing Referential Integrity in SQL

- SQL-92 and SQL-99 support the following options on *deletes* and *updates* on the *parent* relation.
  - Default is **NO ACTION** (delete/update is rejected)
  - **CASCADE** (also delete/update *all tuples* that *refer* to deleted/updated tuple)
  - **SET NULL / SET DEFAULT** (also set *all* tuples that refer to deleted/updated tuple to **NULL/DEFAULT** values)

```
CREATE TABLE Enrolled
(
   -- the sid field default  value is 12345
   sid CHAR(5) DEFAULT 12345,
   uos CHAR(8),
   grade VARCHAR(2),

   PRIMARY KEY (sid,uos),

   FOREIGN KEY (sid) REFERENCES Student
  ON DELETE CASCADE
   -- the on delete cascade conveys
      that an enrolled row should be
      deleted when the student with sid
      that it refers to is deleted

  ON UPDATE SET DEFAULT
  -- the on update set default
      will update the value of sid to a default value
      that is specified as the default
      in this enrolled schema definition
);
```

Table Constraints: Semantic Integrity Constraints

- Examples:
  - "Total marks are between 0 and 100"
  - "Only lecturers of a course can give marks for that course."

- Use SQL **CHECK** constraints, *in-line* like before, or as *separate named* constraints:

  CHECK (semantic-condition)

Example of Semantic Integrity Constraints

```
CREATE TABLE Assessment
(
    sid   INTEGER     REFERENCES Student,
    uos   VARCHAR(8)  REFERENCES UnitOfStudy,
    mark  INTEGER,
   CHECK (mark BETWEEN 0 AND 100)
);
```

SQL: Naming Integrity Constraints

- The **CONSTRAINT** clause can be used to *name* any integrity constraints
- Example:

```
CREATE TABLE Enrolled
(
  sid      INTEGER,
  uos      VARCHAR(8),
  grade  VARCHAR(2),
  CONSTRAINT FK_sid_enrolled   FOREIGN KEY (sid)
                               REFERENCES Student
                               ON DELETE CASCADE,
  CONSTRAINT FK_cid_enrolled   FOREIGN KEY (uos)
                               REFERENCES UnitOfStudy
                               ON DELETE CASCADE,
  CONSTRAINT CK_grade_enrolled CHECK(grade IN ('F',…)),
  CONSTRAINT PK_enrolled   PRIMARY KEY (sid,uos)
);
```

Timing of an Integrity Constraint Check

- Any constraint - *domain, key, foreign-key,* or *semantic* - may be declared as:
  - **NOT DEFERRABLE**
    The default. It means that every time a database modification occurs to tuples that a DBMS sees as being related, the constraint is ***checked immediately***.
  - **DEFERRABLE**
    Gives the option to ***wait until a transaction is complete*** and *then check* the constraint.
    - **INITIALLY DEFERRED** wait until transaction ends
      however, can *dynamically* change this *later* in the transaction by
      SET CONSTRAINTS name **IMMEDIATE**
    - **INITIALLY IMMEDIATE** check immediate,
      however, can *dynamically* change this *later* in the transaction by
      SET CONSTRAINTS name **DEFERRED**

Example of Deferring Constraint Checking

```
CREATE TABLE UnitOfStudy
(
  uos_code       VARCHAR(8),
  title          VARCHAR(20),
  lecturer_id        INTEGER,
  credit_points INTEGER,
  CONSTRAINT UoS_PK PRIMARY KEY (uos_code),
  CONSTRAINT UoS_FK FOREIGN KEY (lecturer_id)
    REFERENCES Lecturer DEFERRABLE INITIALLY DEFERRED
);
```

- Allows us to *insert a new unit of study* referencing a lecturer *who is not present* at *the time of adding a unit of study*, but *who will be added later in the same transaction*.
- Behaviour can be *dynamically changed* within a transaction with the SQL statement:

```
SET CONSTRAINTS UoS_FK IMMEDIATE;
```

Add/Modify/Remove Integrity Constraints*

- Integrity constraints can be *added*, *modified* (*only* applicable to *domain* constraints), and *removed* from an existing schema using **ALTER TABLE** statement

  ALTER TABLE table-name constraint-modification

- where *constraint-modification* is one of the following:

  ADD CONSTRAINT constraint-name new-constraint
  DROP CONSTRAINT constraint-name
  RENAME CONSTRAINT old-name **TO** new-name
  ALTER COLUMN attribute-name domain-constraint

- Example (PostgreSQL syntax):

```
ALTER TABLE Enrolled ALTER COLUMN grade TYPE VARCHAR(3),
                     ALTER COLUMN mark SET NOT NULL;
```

- What happens if the existing data in a table does not fulfil a newly added constraint?

  The constraint **doesn't get created**!
  e.g. "SQL Error: ORA-02296: cannot enable (USER.) - null values found"

Short break:

Let us have some fun again…

THE UNIVERSITY OF SYDNEY

Schema-based Constraints: Assertions

- The integrity constraints seen have so far been defined *within a single table.*
- Some constraints *cannot* be expressed using *only domain constraints* or *referential integrity constraints*; for example:
  - "Every school must have at least five courses offered every semester" – must be expressed as an **assertion**
- Need for ***more general integrity constraints*** that apply **to the *database: schema-based constraints***
  - Integrity constraints that span *several tables*
- Definition of an **Assertion**: *a **predicate** expressing a condition that we wish the **database** to **always satisfy***.
- SQL-92 syntax: **CREATE ASSERTION** assertion-name **CHECK** (condition)
  - When an *assertion* is made, the system *tests* it for its **validity** (i.e., it must *evaluate* to ***true***) on *any update* which *may potentially violate it*
    - Note: This testing may introduce a *significant amount of overhead*; hence assertions should be used with great care.

Assertion Example

- For a **sailing club** to be categorized as ***small***, we require that the **sum** of the ***number of boats*** and ***number of sailors***, ***be less than 10*** at **all times**.

```
CREATE TABLE Sailors (
   sid    INTEGER,
   sname  CHAR(10),
   rating INTEGER,
   PRIMARY KEY (sid),
   CHECK (rating >=1 AND rating <=10),
   CHECK ((SELECT COUNT(s.sid) FROM Sailors s)
          + (SELECT COUNT(b.bid) FROM Boats b) < 10))
);
```

Solution:

```
CREATE ASSERTION smallclub CHECK
((SELECT COUNT(s.sid) FROM Sailors s)
     + (SELECT COUNT(b.bid) FROM Boats b) < 10) );
```

What is *wrong* with the above SQL statement?

The check constraint would be checked only if there is an *update* in the table *Sailors*!

We could keep *adding* boats to be 10 or more and still having *no* constraint *violations*!

$\neg\forall y(R(y) \vee Q(y)) \equiv \exists y(\neg(R(y) \vee Q(y)))$

THE UNIVERSITY OF SYDNEY

Using General Assertions

- Note that *assertions of the form* "**for all X, P(X)**" are *not supported* by SQL and are typically *time-consuming*. Also, note that in *first order logic*:

$$\forall x\, P(x) \equiv \neg\exists x\, (\neg P(x))$$

(**∀** is read "*for all*")

(**∃** is read "*there exists at least one*")

(**≡** means "equivalent", **¬** means "not")

- $\forall x\, P(x)$ reads like: For all values of x the condition P is true.
- $\neg\exists x\, (\neg P(x))$ reads like: There does not exist any value x for which the opposite of condition P is true.

$\neg\forall y(R(y) \lor Q(y)) \equiv \exists y(\neg(R(y) \lor Q(y)))$

THE UNIVERSITY OF SYDNEY

Using General Assertions

- How do we *declare* these types of assertions?
  - Write a query to ***select*** those tuples that ***violate*** the condition **P(x)**, i.e.,such that **¬P(x) is true.**
  - Then use the **NOT EXISTS** clause, i.e., **¬∃x**. This ensures that the **assertion** would return ***true*** whenever the ***returned set*** is ***empty***!
    - The query result ***must therefore be empty*** if the ***assertion is to be satisfied.***
    - If the query result ***is not empty***, the assertion has been ***violated!***

Using General Assertions: Example

Example: Assume we have *four* relations: ***Loan, Borrower, Depositor,*** and ***Account***.

Define an **assertion** that states that *each **loan** has at least one **borrower** who maintains an **account** with a minimum **balance** of $1000.* Assume ***x*** refers to tuples in ***Loan*** *and **y*** refers to tuples in ***Borrower****:*

This query can be written as follows:

$\forall x\ (\exists y\ P(x,y))$

Another way to state this assertion is: *There are **no** loans that have **no** borrowers maintaining an **account** with a minimum **balance** of $1000.*

The above query can be written as

$\neg\exists x\ (\neg\exists y\ (P(x,y)))$

where ***P(x,y)*** = ***(****y.balance >= 1000* ∧ *and x.loan_number = y.loan_number****)*** where the ∧ symbol is read as *"and".*

Using General Assertions: Example

- Using the previous formula: **¬∃x (¬∃y (P(x,y)))**
- **CREATE ASSERTION** balance_constraint **CHECK (*NOT EXISTS*** ¬∃x
  - (**SELECT** * **FROM Loan** ¬∃y

    WHERE *NOT EXISTS*

    (**SELECT Borrower**.loan_number, **Borrower**.balance **FROM Borrower JOIN Depositor**
    ON Borrower.customer_name = Depositor.customer_name

    JOIN Account ON Depositor.account_number = Account.account_number

    WHERE

    Loan.loan_number = Borrower.loan_number

    AND **P(x,y)**

    Account.balance >= 1000))

Using General Assertions: Example

Given the previous assertion: *Each **loan** has at least one **borrower** who maintains an **account** that has a **minimum balance** of $1000.* Assume we have the schema instantiation below.

Assume we wish to **update** the ***Account*** tuple (1221, 1200) with a **withdrawal** of $**700.** If *successful*, this would take the balance down to **$500**.

Borrower

customer_name	address	loan_number
Jane	2, Happy Street	102
John	45, Glad Street	101

Account

account_number	balance
2342	1500
1221	~~1200~~
	500 ?

Depositor

account_number	customer_name
2342	Jane
1221	John

Loan

loan_number	customer_name
101	John
102	Jane

Using General Assertions: Example

The assertion is **checked when** the **update of (1221, 500)** in the table **Account** is attempted.

- Note there that the account number (**1221)** is associated with the borrower **(John).**

- In this example, **loan_number (101)** is associated with *only one* borrower **(John**).

- If we allow the update to proceed, **loan_number (101)** associated with the borrower **(John)** with the account (**1221**) would have a resulting **balance of (500)** which is **less than $1000**, causing a **violation** of the assertion**!** In this case, the **update** is **rejected.**

Using General Assertions: Example

- **Question:** what would the result of the update be if the *loan number of Jane* was **101,** i.e., it is a *joint loan* with John?

- **Answer:** the update would, in this case, *be accepted*!

Borrower

customer_name	address	loan_number
Jane	2, Happy Street	101
John	45, Glad Street	101

Account

account_number	balance
2342	1500
1221	~~1200~~ 500

Depositor

account_number	customer_name
2342	Jane
1221	John

Loan

loan_number	customer_name
101	John
101	Jane

Using General Assertions: Another Example*

*The **sum** of **loan amounts** for each branch **must be less** than* or ***equal** to the **sum** of **all account balances** at the corresponding **branch**.*

Another way to state this assertion is: *There are **no** branches that have the **sum** of **loans <u>greater</u>** than the **sum** of **account balances** at that **branch**.*

```
› CREATE ASSERTION sum_constraint CHECK (NOT EXISTS
  (SELECT * FROM branch
    WHERE (SELECT sum(amount) FROM loan
              WHERE loan.branch_name = branch.branch_name)
               >
              (SELECT sum(amount) FROM account
               WHERE account.branch_name = branch.branch_name)))
```

Using General Assertions

Although **ASSERTION** is in the SQL standard, only a few DBMSs support it. **CHECK** are commonly used as a workaround approach.

Assertion is still an effective design tool to use as the first instance to express constraints at the schema level.

PostgreSQL does not support ASSERTION https://www.postgresql.org/docs/9.2/unsupported-features-sql-standard.html

PostgreSQL does not support subquery in CHECK https://www.postgresql.org/docs/9.1/sql-createtable.html

Outline

- Overview of Integrity Constraints

- Static Integrity Constraints
  - Domain Constraints
  - Key / Referential Constraints
  - Semantic Integrity Constraints
  - Assertions

- **Dynamic Integrity Constraints**
  - Triggers

Triggers

- A **trigger** is a piece of code that is executed *automatically* if some specified *modifications* occur to the database AND a *certain condition* holds true.

- A trigger specification consists of *three* parts:

  ON *event* **IF** *condition* **THEN** *action*

  - ***Event*** (what activates the trigger)
  - ***Condition*** (test the condition's truth to determine whether to execute an action)
  - ***Action*** (what happens if the condition is true)
  - Acronym is **ECA**
  - *Not all triggers have a condition*, i.e., *upon an event an action may be fired.*
  - Triggers are the basis of a specialized type of databases called ***Active Databases****.*

Why Triggers?

- Note that Assertions ***do not*** *modify* the data:
  - Need a more powerful mechanism to *check conditions* and *modify* the data in a database.
- Examples of use of triggers:
  - **Constraint maintenance**
    - Triggers can be used to *maintain* foreign-key and semantic constraints; commonly used with ON DELETE and ON UPDATE
  - **Business rules**
    - Dynamic business rules can be encoded as triggers. E.g., rules to adjust discounts based on stock availability
  - **Monitoring**
    - E.g., to react on the insertion of some sensor readings.
  - **Auditing/Compliance requirements**
    - E.g., government reporting regulations

Trigger Example (SQL:1999)

Example: If the sum of marks of all current assessments for a student is greater than or equal to 50, enter the grade "P".

Event: *new or updated assessment*

Condition: *check whether the sum of marks is greater than or equal to 50*

Action: *Enter grade "P" if the* ***condition*** *evaluates to true*

```
CREATE TRIGGER gradeEntry
  AFTER INSERT OR UPDATE ON Assessment        Event
  BEGIN
    UPDATE Enrolled E                         Action
      SET grade='P'
     WHERE ( SELECT SUM(mark)
             FROM Assessment A                Condition
            WHERE A.sid=E.sid AND
                 A.uos=E.uosCode ) >= 50;
 END;
```

Triggering Events and Actions

- *Triggering* event can be **INSERT**, **DELETE** or **UPDATE**
- Triggers on **update** can be **restricted** to **specific attributes**

  CREATE TRIGGER overdraft-trigger

  AFTER UPDATE OF *balance* **ON** Account

- Values of attributes **before** and **after** an update can be *referenced:*
  - **REFERENCING OLD ROW AS** name: for *deletes* and *updates*
  - **REFERENCING NEW ROW AS** name: for *inserts* and *updates*
  - In PostgreSQL: separate **OLD** and **NEW** variable automatically generated with a trigger function (PL/pgsql).

Trigger Granularity

- **Granularity**
  - ***Row-level trigger***: A **row-level trigger** is fired for **each row** that needs to be updated.
  - ***Statement-level trigger***: A **statement trigger** fires **once per triggering event** and *regardless* of *how many rows are modified* by the insert, update, or delete event.

- *Statement-level* trigger is usually more *efficient* when dealing with SQL statements that update *many rows as a group:*
  - if several rows in a table are updated, then a *statement-level* trigger would, in this case, be executed **only once.**

Example of Row Level Trigger

Example: Assume that for auditing purposes, we define **a trigger** that ensures that any time a row is *inserted* or *updated* in the *employee* table, *the current user name of the person that made the modification* and the *time,* are *timestamped into the row*. It also checks that an *employee's name* is *provided* and that *the salary* is a *positive* value.

```
CREATE TABLE emp (
    empname text,
    salary integer,
    last_date timestamp,
    last_user text
);
```

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
    BEGIN
        -- Check that empname and salary are given
        IF NEW.empname IS NULL THEN
            RAISE EXCEPTION 'empname cannot be null';
        END IF;
        IF NEW.salary IS NULL THEN
            RAISE EXCEPTION '% cannot have null salary', NEW.empname;
        END IF;

        -- check if salary is negative
        IF NEW.salary < 0 THEN
            RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
        END IF;

        -- Remember who changed the payroll when
        NEW.last_date := current_timestamp;
        NEW.last_user := current_user;
        RETURN NEW;
    END;
 $emp_stamp$ LANGUAGE plpgsql;
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Trigger levels - Syntax

- Instead of executing a separate action for each affected row, **a single action** can be executed **for all rows** affected by **a statement**

  - Use **FOR EACH STATEMENT** (this is the **default**) instead of **FOR EACH ROW**

- Statement-level triggers are more efficient when dealing with SQL statements that *update many rows as a group*.

Example of Statement-Level Triggers*

Example: Assume the average salary is *timestamped* in a *dedicated relation* every time new employees are inserted or employee records are updated

```
CREATE FUNCTION Salary_Average() RETURNS trigger AS $$
          BEGIN
          INSERT INTO salaryaverages(datestamp,average) VALUES
          (CURRENT_DATE, (SELECT AVG(salary) FROM Employee));
          RETURN null;
  END;
 $$ LANGUAGE plpgsql;
 CREATE TRIGGER RecordNewAverage
  AFTER UPDATE OF Salary or INSERT ON Employee
  FOR EACH STATEMENT
    EXECUTE PROCEDURE Salary_Average();
```

Triggers

```
CREATE [OR REPLACE] TRIGGER trigger-name
   { BEFORE     } { INSERT            }
   { AFTER      } { DELETE            }  ON table-name
   { INSTEAD OF } { UPDATE OF attr    }

   REFERENCING   { OLD }  TABLE AS variable-name--optional
                 { NEW }

   FOR EACH ROW                -- optional; otherwise, a statement trigger
   WHEN (condition)            -- optional
   DECLARE
       <local variable declarations>
   BEGIN
     <PL/SQL block>
   END;
```

In PostgreSQL, this is replaced by a trigger procedure

PostgreSQL trigger procedure: https://www.postgresql.org/docs/current/sql-createtrigger.html

Things to consider

- Things to consider when deciding to use a **row** or **statement** level trigger: **Update Cost**
  - How many rows are updated?
  - How often is a ***row-level*** trigger executed?
  - How often is a ***statement-level*** trigger executed?

THE UNIVERSITY OF SYDNEY

Statement vs Row Level Trigger: A Recap

Row Level Triggers vs.	**Statement Level Triggers**
Row level triggers executes once for each and every row that is updated/inserted/deleted.	Statement level triggers executes only once for each single SQL statement.
Mostly used for data auditing purpose.	Used for bulk modifications performed on the table.
“FOR EACH ROW” clause is present in CREATE TRIGGER command.	“FOR EACH ROW” clause is omitted in CREATE TRIGGER command.
Example: If 1500 rows are to be inserted into a table, the row level trigger would execute 1500 times.	Example: If 1500 rows are to be inserted into a table in one single statement, the statement level trigger would execute only once.

Some Tips on Triggers

- Use BEFORE triggers
  - Usually for *checking integrity constraints*
- Use AFTER triggers
  - Usually for *integrity maintenance and update propagation*

You Should Now be Able To

- Capture Integrity Constraints in an SQL Schema
  - Including key constraints, referential integrity, domain constraints and semantic constraints
- Formulate complex semantic constraints using Assertions
- Know when to use Assertions and CHECK constraints
- Know the semantic of deferring integrity constraints
- Be able to formulate simple triggers
  - Know the difference between row-level & statement-level triggers

References

- Ramakrishnan/Gehrke (3rd edition - the ‘Cow’ book)
  - **Sections 3.2-3.3 and Sections 5.7-5.9**
  - *Integrity constraints are covered in different parts of the SQL discussion; only brief on triggers*
- Kifer/Bernstein/Lewis (2nd edition)
  - Sections 3.2.2-3.3 and Chapter 7
  - *Integrity constraints are covered as part of the relational model, but there is a good dedicated chapter (Chap 7) on triggers*
- Ullman/Widom (3rd edition)
  - Chapter 7
  - *Has a complete chapter dedicated to both integrity constraints & triggers.*
- Michael v. Mannino: “Database - Design, Application Development and Administration”
  - *Includes a good introduction to triggers.*

Next Week

- Advanced SQL
  - Nested Queries
  - Aggregation & Grouping
  - NULL Values

- Readings:
  - **Ramakrishnan/Gehrke (Cow book), Chapter 5 & Chapter 4.2.5**
  - Kifer/Bernstein/Lewis book, Chapter 5 & 3.2-3.3
  - Ullman/Widom, Chapter 6

See you next time!