COMP9120

Week 5: Database Integrity

Semester 1, 2022

Professor Athman Bouguettaya
School of Computer Science

THE UNIVERSITY OF SYDNEY

Contents developed by Dr Lijun Chang

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 Darug people and pay my respects to their Elders, past, present and emerging.

I further acknowledge the Traditional Owners of the country on which you are on and pay respects to their Elders, past, present and future.

Copyright warning

Outline

- **Overview of Integrity Constraints**

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

- **Dynamic Integrity Constraints**
  - Triggers

- **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** *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 (*eventually*) satisfies all specified ICs.
  - Not *necessarily at all times.*

Integrity Constraints (ICs)

› Example 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 be higher than 100.

Automatically Capturing ICs

- Why do we need to capture integrity constraints:
  - 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).
  - Stored data is more faithful to the real-world meaning (semantics) of the domain application
    - Avoid data entry errors (e.g., inserting a *grade* into the **Student** table which does not exist).
  - Easier application development and better maintainability because ICs are centrally managed by the DBMS.
    - We do not have to worry about ***how*** integrity constraints are enforced/implemented.

Integrity Constraints in a Database

- ICs are *specified* as part of the database *schema design*
  - The database designer is responsible for ensuring that the integrity constraints *do not contradict* each other!
    - Detection process could be automated but this may introduce unacceptable overhead.
- ICs are *checked* when the related parts of the database are *modified.* However
  - Can specify when ICs should be checked: after a SQL statement, or at the end of a 'transaction'
    - Transaction: a **group of statements** to be executed atomically
      (will look at "ACID" properties of transactions later in the semester)
- Possible *ways to react* if an IC is violated:
  - Reject database operation
  - *Abort* the 'transaction' – *rollback* operations which are part of the current 'transaction'
  - Execution of *"maintenance" operations* to make DB legal again

THE UNIVERSITY OF SYDNEY

An Informal Introduction to Transaction

- Transactions: A group of statements to be executed **atomically**

  BEGIN;
  A group of SQL statements;
  COMMIT;

- A SQL statement usually starts with the following keywords and ends with a semicolon
  - **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;

Types of Integrity Constraints

- ***Static*** **Integrity Constraints**

  They describe conditions that every *legal instance* of a database must satisfy, i.e., these 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*

- **Dynamic Integrity Constraints**

  They are predicates on *database state changes* that capture conditions *over two or more states* and therefore 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 inserted in the database
  - Also: queries are tested to ensure that the comparisons make sense.
  - Most simply, each attribute needs to have a *data type*

- SQL DDL allows domains of attributes to be further restricted in the **CREATE TABLE** statement with 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 values for 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 with an in-line check constraint

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

- Examples:
  - Gender can be ‘male’ or ‘female’

    gender **VARCHAR**(6*)* **CHECK(** gender **IN** (‘male’, ‘female’) **)**
  - 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 DEFAULT 'P' CHECK(VALUE IN
                                    ('F','P','C','D','H'))
```

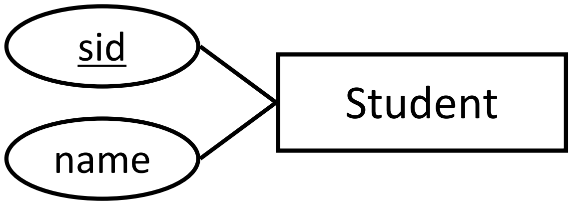
equivalent to:

```
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 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)
);
```

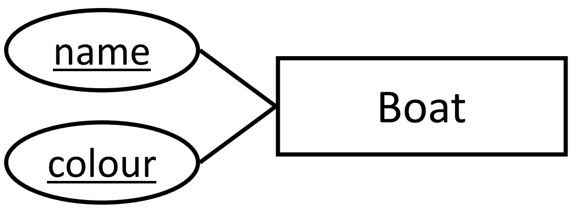
› A primary key is automatically **UNIQUE** and **NOT NULL**

- 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 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

› **Referential Integrity**: for each tuple in the referring relation whose foreign key value is α, there must be a tuple in the referred relation whose value of the referred attribute is also α

- e.g. Enrolled(*sid*: integer, ucode: string, semester: string)
  sid is a foreign key referring to Student:
- If all foreign key constraints are enforced, referential integrity is achieved, i.e., no dangling references

Foreign Keys in SQL

› Only students listed in the Students 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
);
```

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

Enforcing Referential Integrity in SQL

- SQL-92 and SQL-1999 support all 4 options on deletes and updates.
  - Default is **NO ACTION** (delete/update is rejected)
  - **CASCADE** (also delete/update all tuples that refer to deleted/updated tuple)
  - **SET NULL / SET DEFAULT** (sets foreign key value of referencing tuple)

```
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
   -- the on delete cascade conveys
      that an enrolled row should be
      deleted when the student with sid
      that it refers to is deleted
   ON DELETE CASCADE

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

Semantic Integrity Constraints (Table 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)
);
```

Deferring Constraint Checking

- Any constraint - domain, key, foreign-key, semantic - may be declared:
  - **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 afterwards.

  - **DEFERRABLE**
    Gives the option to wait until a transaction is complete before checking the constraint.
    - **INITIALLY DEFERRED** wait until transaction end,
      but allow to dynamically change later by
      SET CONSTRAINT name **IMMEDIATE**
    - **INITIALLY IMMEDIATE** check immediate,
      but allow to dynamically change later by
      SET CONSTRAINT name **DEFERRED**

Example of Deferring Constraint Checking

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

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

```
SET CONSTRAINT UoS_FK IMMEDIATE;
```

Add/Modify/Remove Integrity Constraints

- Integrity constraints can be added, modified (applicable *only* 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:

  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?

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

Short 5 mn break:

please stand up, stretch, and move around

Assertions

- The integrity constraints seen so far *are defined within a single table.*
- Some constraints cannot be expressed by 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
  - E.g. integrity constraints over several tables
- **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 (must evaluate to true), and tests it again on every update that may 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 must be less than or equal to 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))

);
```

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

Using General Assertions

- Note that *assertions of the form* “for all X, P(X)” are not supported by SQL.
- How to declare assertions?
  - Write a query that highlights (selects) the tuples that ***violate*** the condition.
  - Then use the NOT EXISTS clause to make sure the assertion yields true whenever the tuple set returned by the query is empty.
- Query result *must be empty to be true*. 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 a constraint that *every loan has at least one borrower who maintains an account with a minimum balance of $1000.00*

- **CREATE ASSERTION** balance_constraint **CHECK (NOT EXISTS**
  - (**SELECT** * **FROM** loan
    - **WHERE NOT EXISTS**
    - (**SELECT** * **FROM** borrower, depositor, account
    - **WHERE** loan.loan_number = borrower.loan_number
    - **AND** borrower.customer_name = depositor.customer_name
    - **AND** depositor.account_number = account.account_number
    - **AND** account.balance >= 1000))

Using General Assertions: Another Example

Example: *The sum of loan amounts for each branch must be less than the sum of all account balances at the 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: Another Example

- Example: Assume we have three relations: ***student, course,*** *and* ***takes***. For each tuple in the ***student*** relation, the value of the attribute *tot_cred* must equal the sum of credits of courses that the student has completed successfully.
- **CREATE ASSERTION** credits_earned_constraint
- **CHECK (NOT EXISTS**
  - (**SELECT** ID
    - **FROM** student
    - **WHERE** tot-cred <> (**SELECT** sum(credits)
      - **FROM** takes **NATURAL JOIN** course
      - **WHERE** student.ID=takes.ID
      - **AND** grade **IS NOT NULL**
      - **AND** grade <> ‘F’)))

Using General Assertions

Although **ASSERTION** is in the SQL standard, only a few DBMSs support it (e.g. Oracle). **CHECK** is commonly used as a workaround approach.

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 section 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)

  - 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****.* They are also referred to as *Rule-based Databases.*

Why Triggers?

- Assertions *cannot modify* the data. They are not associated to any specific tables or events in the database
  - Need a more powerful mechanism to *check conditions* and *modify* the data in a database.
- 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
- Monitoring
  - E.g. to react on the insertion of some sensor readings into the database.
- Auditing
  - Compliance requirements

Trigger Example (SQL:1999)

Example: *If the sum of marks of all current assessments for a students 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
  BEGIN
   UPDATE Enrolled E
     SET grade='P'
    WHERE ( SELECT SUM(mark)
           FROM Assessment A
          WHERE A.sid=E.sid AND
              A.uos=E.uosCode ) >= 50;
 END;
```

THE UNIVERSITY OF SYDNEY

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:*
  - if several rows in a table are updated, then a *statement-level* trigger would in this case be executed **only once.**

Row Level Trigger

For auditing purposes, this example trigger ensures that any time a row is inserted or updated in the table, the current user name and time are timestamped into the row. It also checks that an employee's name is given 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** instead of **FOR EACH ROW** (this is the **default**)

- Statement-level triggers can be more efficient when dealing with SQL statements that update many rows…

Statement-Level Triggers

The average salary is timestamped 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/9.5/static/plpgsql-trigger.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?

Statement vs Row Level Trigger

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, 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

- Good overviews:
  - Ramakrishnan – Brief overview Section 5.8, 5.9
  - Kifer/Bernstein/Lewis: “Database Systems - An Application-oriented Approach”, 2nd edition, Chapter 7.
  - Michael v. Mannino: “Database - Design, Application Development and Administration”

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 a good dedicated chapter (Chap 7) on triggers*
- Ullman/Widom (3rd edition)
  - Chapter 7
  - *Has a complete chapter dedicated to both integrity constraints&triggers. Good.*
- Michael v.Mannino: “Database - Design, Application Development and Administration”
  - *Include a good introduction to triggers.*

Next Week

- **Transaction Management**
  - Transaction Concept
  - Serializability

- Readings:
  - **Ramakrishnan/Gehrke (Cow book), Chapter 16**
  - Kifer/Bernstein/Lewis book, Chapter 18
  - Ullman/Widom, Chapter 6.6 onwards

See you next week!