Lecture 10

Database Construction (and Usage)

More on Modifications and Table Creation
Assertions
Triggers

Summary – Modifications

- Modifying the contents of a database:
  - Insertions
    ```
    INSERT INTO tablename VALUES tuple
    ```
  - Deletions
    ```
    DELETE FROM tablename WHERE test over rows
    ```
  - Updates
    ```
    UPDATE tablename
    SET    attribute = value
    WHERE  test over rows
    ```

Insertions with queries

- The values to be inserted could be taken from the result of a query:

  ```
  INSERT INTO tablename (query)
  ```

  – Example:

```
INSERT INTO GivenCourses
 (SELECT course, period + 2, teacher, NULL
  FROM   GivenCourses
  WHERE  period <= 2);
```

All courses that are given in periods one and two are also scheduled to be given two periods later, with the same teacher.

Explicit attribute lists

- Attribute order could be given explicit when inserting.
  – Example:

```
INSERT INTO
 GivenCourses(course, period, teacher, nrStudents)
 (SELECT course, period + 2, teacher, NULL
  FROM   GivenCourses
  WHERE  period <= 2);
```

Perhaps the teacher and nrStudents attributes were listed in the other order in the definition of the table? Doesn't matter anymore since they are explicitly listed.

Quiz

What will the following insertion result in?

course	period	teacher	numStud

```
INSERT INTO
 GivenCourses(course, period, teacher)
 VALUES ('TDA357', 3, 'Mickey');
```

– Attribute lists can be partial. Any attributes not mentioned will be given the value a default value, which by default is NULL.

Default values

- Attributes can be given default values.
  – Specified when a table is defined using the DEFAULT keyword.
  – Example:

```
CREATE TABLE GivenCourses (
  course      CHAR(6),
  period      INT,
  teacher     VARCHAR(50),
  nrStudents INT DEFAULT 0,
  … constraints …
);
```

  – Default default value is NULL.

Insertion with default values

- Leaving out an attribute in an insertion with explicitly named attributes gives that row the default value for that attribute:

```
INSERT INTO
 GivenCourses(course, period, teacher)
 VALUES ('TDA357', 3, 'Mickey');
```

- When no attribute list is given, the same effect can be achieved using the DEFAULT keyword:

```
INSERT INTO GivenCourses
 VALUES ('TDA357', 3, 'Mickey', DEFAULT);
```

Quiz!

Courses

code	*name*
TDA357	Databases
TIN090	Algorithms

GivenCourses

course	*per*	*teacher*	*nrSt*
TDA357	2	Mickey	130
TDA357	4	Tweety	95
TIN090	1	Pluto	62

```
DELETE FROM Courses
WHERE  code = 'TDA357';
```

Error, because of the reference from GivenCourses to Courses. Is this reasonable?

Policies for updates and deletions

- Rejecting a deletion or update in the presence of a reference isn't always the best option.
- SQL provides two other methods to resolve the problem: Cascading or Set NULL.
  - Default is RESTRICT: reject the deletion/update.

Cascading

- Cascading: When the referenced row is deleted/updated, also delete/update any rows that refer to it.
  - Typically used for "parts of a whole".
  - Set using ON [DELETE|UPDATE] CASCADE

```
CREATE TABLE GivenCourses (
  course CHAR(6),
  CONSTRAINT CourseExists
    FOREIGN KEY course REFERENCES Courses(code)
      ON DELETE CASCADE
      ON UPDATE CASCADE
  … more columns and constraints …
);
```

Set NULL

- Set NULL: When the referenced row is deleted/updated, set the corresponding attribute in any referencing rows to NULL.
  - Typically used when there is a connection, but one that does not affect the actual existence of the referencing row.
  - Set using ON [DELETE|UPDATE] SET NULL

```
CREATE TABLE GivenCourses (
  teacher VARCHAR(50),
  CONSTRAINT TeacherExists
    FOREIGN KEY teacher REFERENCES Teachers(name)
      ON DELETE SET NULL
      ON UPDATE CASCADE
  … more columns and constraints …
);
```

Quiz!

Argue for sensible policies for deletions and updates for the Lectures table.

```
Lectures(course, period, weekday, hour, room)
  (course, period) -> GivenCourses.(course, period)
  room -> Rooms.name
```

- GivenCourses.(course, period):
  - ON DELETE: What?
  - ON UPDATE: What?
- Rooms.name:
  - ON DELETE: What?
  - ON UPDATE: What?

Quiz!

Argue for sensible policies for deletions and updates for the Lectures table.

```
Lectures(course, period, weekday, hour, room)
  (course, period) -> GivenCourses.(course, period)
  room -> Rooms.name
```

- GivenCourses.(course, period):
  - ON DELETE CASCADE
  - ON UPDATE CASCADE or RESTRICT
- Rooms.name:
  - ON DELETE SET NULL or RESTRICT
  - ON UPDATE CASCADE

Single-attribute constraints

- Many constraints affect only the values of a single attribute. SQL allows us to specify such constraints together with the attribute itself, as *inline constraints.*

```
CREATE TABLE Courses (
  code CHAR(6) CONSTRAINT CourseCode PRIMARY KEY,
  name VARCHAR(50)
);
```

- More than one inline constraint on the same attribute is fine, just put them after one another.
- Default values should be specified before constraints.

Special case: NOT NULL

- Specifying that a value must be non-NULL can be done with a simplified syntax:

```
CREATE TABLE Courses (
  code CHAR(6) CONSTRAINT CourseCode PRIMARY KEY,
  name VARCHAR(50) NOT NULL
);
```

instead of

```
CREATE TABLE Courses (
  code CHAR(6) CONSTRAINT CourseCode PRIMARY KEY,
  name VARCHAR(50) CHECK (name IS NOT NULL)
);
```

Special case: REFERENCES

- When a foreign key constraint is defined inline, the FOREIGN KEY keywords can be left out.
- An attribute that references another attribute could be seen as holding copies of that other attribute. Why specify the type again?

```
CREATE TABLE GivenCourses (
  course REFERENCES Courses(code),
  … more columns and constraints …
);
```

  - The type can be left out even if the foreign key constraint is specified separately.

Quiz!

It might be tempting to write

```
CREATE TABLE GivenCourses (
  course REFERENCES Courses(code) PRIMARY KEY,
  period INT CHECK (period IN (1,2,3,4)) PRIMARY KEY,
  … more columns and constraints …
);
```

Why will this not work?

An inline constraint only constrains the current attribute. What the above tries to achieve is to declare two separate primary keys, which is not allowed in a table.

Constraints

- We have different kinds of constraints:
  - Dependency constraints ($X \rightarrow A$)
    - Table structure, PRIMARY KEY, UNIQUE
  - Referential constraints
    - FOREIGN KEY … REFERENCES
  - Value constraints
    - CHECK
  - Miscellaneous constraints (like multiplicity)
    - E.g. no teacher may hold more than 2 courses at the same time.
    - How do we handle these?

Quiz!

”No teacher may hold more than two courses in the same period!”

How can we formulate this constraint in SQL?

```
NOT EXISTS (
 SELECT   teacher, period
 FROM     GivenCourses
 GROUP BY teacher, period
 HAVING   COUNT(course) > 2
);
```

course	period	teacher	numStud

Assertions

- Assertions are a way to specify global constraints on a database.
  - Create using CREATE ASSERTION:

```
CREATE ASSERTION name CHECK test
```

  - Example:

```
CREATE ASSERTION NoT
CHECK (NOT EXISTS
        (SELECT   te
         FROM     GivenCourses
         GROUP BY teacher, period
         HAVING   COUNT(course) > 2)
);
```

PostgreSQL does not support CREATE ASSERTION, So we emulate them using TRIGGER

Triggers

- When something wants to change the database in some way, trigger another action as well or instead.
  - Example (silly): Whenever a new course is inserted in Courses, schedule that course to be given in period 1, with NULL for the teacher and nrStudents fields.
  - Example: Whenever a lecture is scheduled to take place at 8:00, schedule the lecture to 10:00 instead.

Assertions as triggers

- ”Instead” could mean to do nothing, i.e. reject the update, which means we can use triggers to simulate assertions.
  - Still costly, but puts the burden on the user to specify when the conditions should be checked (hand optimization).
  - Example: Whenever a teacher is scheduled to hold a course in a period where he or she already holds two courses, reject the insertion.

Basic trigger structure

```
CREATE TRIGGER name
 [BEFORE|AFTER] [INSERT|DELETE|UPDATE] ON tablename
 FOR EACH [ROW|STATEMENT]
 WHEN condition
 EXECUTE PROCEDURE function()
```

Decide whether to run the trigger or not.

What should happen when the trigger is triggered.

A trigger is sometimes referred to as an Event-Condition-Action rule (or ECA rule)

Stored procedures

```
CREATE FUNCTION name() RETURNS TRIGGER AS $$
BEGIN
    <statements>
END
$$ LANGUAGE 'plpgsql';
```

We only consider TRIGGER procedures

Where **statement** is

-
```
IF (condition)
     THEN statement
     ELSE statement
END IF;
```
- `RAISE EXCEPTION 'message';`
- `sqlstatement;`

Example trigger:

```
CREATE FUNCTION addDefaultGivenCourse() RETURNS
TRIGGER AS $$
BEGIN
    INSERT INTO GivenCourses(course, period)
      VALUES (NEW.code, 1);
END
$$ LANGUAGE 'plgsql';
```

'NEW' refers to the newly inserted tuple

```
CREATE TRIGGER DefaultScheduling
 AFTER INSERT ON Courses
 FOR EACH ROW
 EXECUTE PROCEDURE addDefaultGivenCourse();
```

Out shorthand notation:

```
CREATE FUNCTION addDefaultGivenCourse() RETURNS
TRIGGER AS $$
BEGIN
    INSERT INTO GivenCourses(course, period)
      VALUES (NEW.code, 1);
END
$$ LANGUAGE 'plgsql';
```

```
addDefaultGivenCourse() →
      INSERT INTO GivenCourses(course, period)
      VALUES (NEW.code, 1);
```

Trigger events

- The event clause of a trigger definition defines when to try the trigger:
  - AFTER or BEFORE (for tables)
  - INSERT, DELETE or UPDATE
    - An update could be an UPDATE OF (attributes) to make it consider only certain attributes.
  - ON which table to apply the trigger.
  - Example: `AFTER INSERT ON Courses`

```
CREATE TRIGGER name
 event clause
 "for each" clause
 condition clause
 EXECUTE PROCEDURE x()
```

FOR EACH ROW

- A single insert, update or deletion statement could affect more than one row.
- If FOR EACH ROW is specified, the trigger is run once for each row affected, otherwise once for each statement.
- Default is FOR EACH STATEMENT, which could also be stated explicitly.

```
CREATE TRIGGER name
 event clause
 "for each" clause
 condition clause
 EXECUTE PROCEDURE x()
```

Trigger Condition

- The condition specifies whether the action should be run or not.
- Any boolean-valued expression may be used.
- Evaluated before or after the event, depending on BEFORE or AFTER.
- Can refer to the NEW and OLD rows

```
CREATE TRIGGER name
 event clause
 "for each" clause
 condition clause
 EXECUTE PROCEDURE x()
```

Example:

```
WHEN
  (NEW.code
    LIKE 'TDA%')
```

Example revisited

```
f() → INSERT INTO GivenCourses(course, period)
      VALUES (NEW.code, 1);

CREATE TRIGGER DefaultScheduling
 AFTER INSERT ON Courses
 FOR EACH ROW
 EXECUTE PROCEDURE f();
```

Why must this be run AFTER INSERT? Why not BEFORE?

Why?

Example revisited

```
f() → INSERT INTO GivenCourses(course, period)
      VALUES (NEW.code, 1);

CREATE TRIGGER DefaultScheduling
 AFTER INSERT ON Courses
 FOR EACH ROW
 EXECUTE PROCEDURE f();
```

Why must this be run AFTER INSERT? Why not BEFORE?

Because there is a foreign key constraint from GivenCourses to Courses, and until we have inserted the row into Courses, there would be nothing for the new row in GivenCourses to refer to.

Recap on views

- Views are persistent named queries – they can be referred to just as if they were tables, but their data is contained in other (base) tables.
- Also referred to as *virtual tables*.

```
CREATE VIEW DBLectures AS
 SELECT room, hour, weekday
 FROM   Lectures
 WHERE  course = 'TDA357'
        AND period = 3;
```

Updating views

- Views contain no data of their own, and so cannot normally be updated.
- But views can be queried without containing any data of their own. The trick is to translate the query on the view into what it really means, i.e. the view definition.
- Why not do the same for modifications?

Triggers on views

- We can define what modifications on views mean using triggers.
- Special form of event for views only: INSTEAD OF.

```
f() → INSERT INTO Lectures
      VALUES ('TDA357', 2, NEW.weekday,
                NEW.hour, NEW.room);

CREATE TRIGGER DBLectureInsert
 INSTEAD OF INSERT ON DBLectures
 FOR EACH ROW
 EXECUTE PROCEDURE f()
```

Summary – Triggers

- Triggers specify extra actions to take on certain events.
  - Event: BEFORE or AFTER a modification
  - Condition: test if we should run the trigger
  - Action: The stuff to be done.
    - SET to change values in the rows being modified.
- Triggers can be defined on views
  - Event: INSTEAD OF

Next time, Lecture 11