TDA357/DIT621 – Databases

Lecture 8 – Functions and triggers

Jonas Duregård

Last two weeks: high-level design

- Both ER-diagrams and Functional Dependencies allow us to model a domain
- "courseCode -> teacher" is a statement with at least three interpretations:
  - Formal: "Rows that agree on courseCode must also agree on teacher"
  - Informal: "If I know a course code, I can find its teacher"
  - Domain conceptual: "At most one teacher can work on a course"
- ER is generally more expressive than FD's (we can say more about the domain using ER), but there are some exceptions (things FD's can do that ER can't)

Today: Triggers and stuff

Here are a few things that our high-level design can not express:

- Events
  - Example from domain: "when a student unregisters, the first student from the waiting list should be moved …"
  - ER describes what a database can contain, not operations on it
  - The design can only make sure to accommodate both the state before the event and after the event, but not the event itself
- Advanced cross-table constraints
  - Example from domain "A student can only register if they have passed all prerequisites"
  - Can not be expressed using reference/unique/value constraints

Two ways of solving the problem

- Bad way: Use application logic
  - Make sure that every application that uses the database correctly performs all actions
  - This exposes operations that can corrupt the database ☹

- Better way: Move as much logic as possible into the DBMS
  - Expose only safe operations
  - Database consistency is guaranteed
  - Implement only once even if used by multiple applications
  - Compare to hiding unsafe internal operations in OOP

Compare to Java

- Would this be an acceptable java interface?

```
/* Unregister a student. After calling this method you
must always call checkFree(code)&&hasWaiting(code) and
if true you must call register(code,popWaiting(code,1)),
followed by compactWL(code) otherwise the database will
collapse, locust will descend on your crops and blood
shall rain from the sky. */
public void unregisterStudent(String id, String code){
  ...
}
```

- No! If someone asks you to use code like this, never speak to them again.
- So why should we accept this from a database interface?

Side note: atomicity

- All SQL modifications are atomic: If I execute a DELETE/UPDATE/INSERT and there is an error, no rows will have changed
  - For instance, for a delete, if any matching row can not be deleted, nothing gets deleted
  - Intermediate changes are never visible to other users of the database (rows do not disappear and then reappear again, they either change or they don't)
  - This says something about how intricate a DBMS is, imagine implementing this on a data structure in Java! (A method that removes all members satisfying a criteria, but rolls back all changes if there is an error)
  - Works even if the server loses power in the middle of an update(!) (The update will either be performed completely, or nothing is changed)

Cascading

- Remember: deletes/updates may fail due to references from other tables
  - I can not remove a student unless I first remove all that students' grades
- By default, a query that attempts to delete a referenced row fails and nothing is deleted, this can be changed when creating the reference:
- Delete all referencing rows as well:
  ```
  student TEXT REFERENCES Students(id) ON DELETE CASCADE
  ```
  - Can potentially lead to deleting the whole database

Modes for ON DELETE/UPDATE

- ON [DELETE/UPDATE] [CASCADE/SET NULL /RESTRICT/NO ACTION]
  - ON DELETE CASCADE: Delete this row if the referenced value is deleted
  - ON UPDATE CASCADE: Update this value if the referenced value is updated
  - SET NULL: Set this value to NULL if the referenced value is updated/deleted
  - RESTRICT and NO ACTION: Raise an error, NO ACTION is the default behavior
- ON UPDATE is usually OK to CASCADE, for ON DELETE, be more careful

A complete example

- Here is the LimitedCourses table, extended with the follow behavior:
  - If a course is deleted (in Courses), its limit is also deleted
  - If a course code is changed in Courses, it is also changed here
- Both of these make sense: Why give an error when deleting a course simply because it has a limitation?

```
CREATE TABLE LimitedCourses(
   course CHAR(6) PRIMARY KEY,
   seats INTEGER NOT NULL,
   FOREIGN KEY (course) REFERENCES Courses
      ON DELETE CASCADE ON UPDATE CASCADE
);
```

- Note that this says nothing about the other direction (if a course code is changed in LimitedCourses for instance)

What is the sensible ON UPDATE/DELETE here?

- Argue for a sensible policy for the references in Lectures:

```
Lectures(course, year, weekday, hour, room)
    (course, year) -> Courses.(code, year)
    room -> Rooms.name
```

- For Courses.(code, year):
  ON DELETE CASCADE ON UPDATE (CASCADE/nothing)
  If a course is deleted, delete its lectures, if a year is changed…?
- For Rooms.name:
  ON UPDATE CASCADE ON DELETE (nothing/SET NULL)
  If a room is renamed, update lectures
  If a used room is removed, give an error (or set to NULL? Needs to allow null values for room attribute)
- The default of raising an error is never totally wrong (it will never corrupt your database, but may make it less usable)

How to identify ON DELETE/ON UPDATE

- If you need to do something like "When a student is deleted, it should automatically be unregistered from all courses"
  - Note how this is clearly not something that we can model in ER

Advanced cross-row/cross-table constraints

- Often, we want constraints affecting multiple rows/multiple tables
- Some such constraints can already be implemented:
  - Unique/primary key constraints are cross-row constraints
    (You need to look across rows to know there are no conflicts)
  - Foreign keys are cross-table constraints
    (You need to look in another table to know if a value is allowed)
- Other constraints can not be expressed using references/keys, examples:
  - Two columns should be mutually exclusive ("shared keys")
  - A column should contain consecutive numbers starting from 1
  - …

ASSERTIONS

- Assertions are part of the SQL standard
- They allow us to write conditions that should be globally true for the database
- Syntax:

```
CREATE ASSERTION <assertion name> AS
   CHECK <condition>;
```

- Very difficult to implement efficiently in a DBMS
- For instance: We can write an assertion that states that all course registrations have happened within the last year from today
  - When should this be checked? What happens when it is suddenly false?
- Assertions are not implemented in Postgres, or in most major DBMS

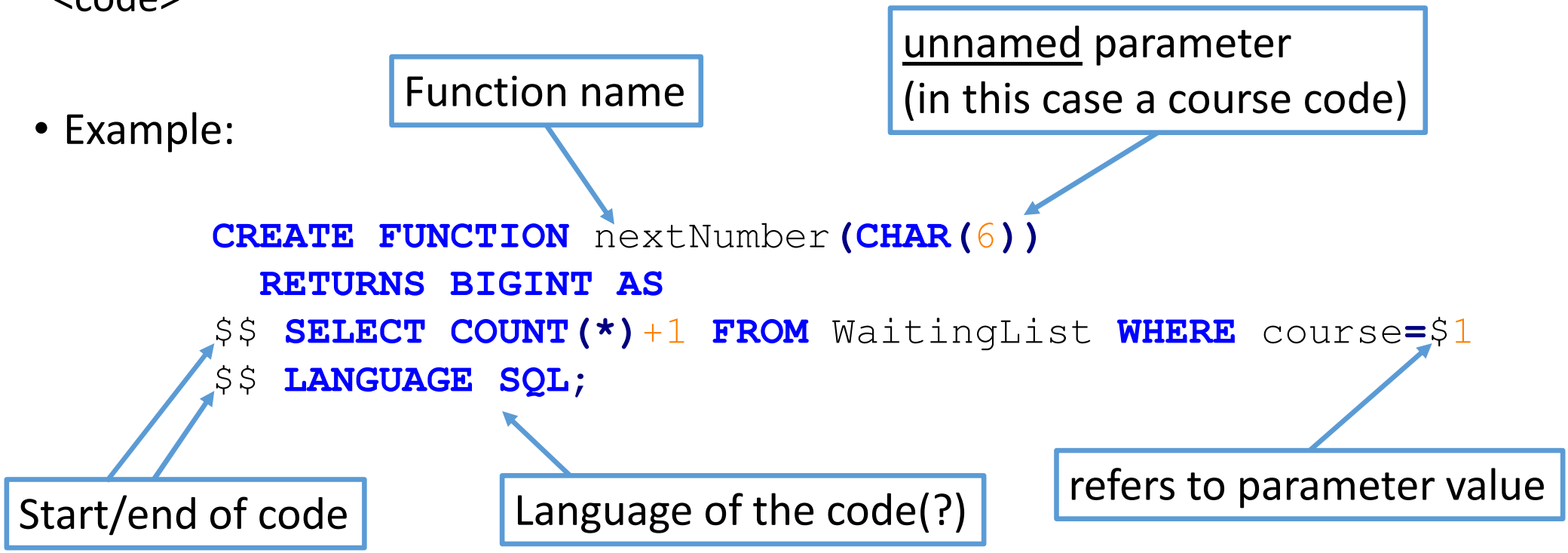
User created functions

- *Most* DBMS allows users to create functions, these can be used similar to COALESCE() and other functions we have seen
- Functions are stored in the DB server and executed in queries etc.
  - Sometimes called "stored procedures"
- Reasons not to use functions
  - They do not fit nicely into the relational data model
  - May be poorly optimized
  - Poorly standardized between DBMSs (code from one DBMS may work in another with some alterations, or it may not work at all)
- Reasons to use functions
  - There are some things that simply cannot be done without them

CREATE FUNCTION in Postgres

CREATE FUNCTION <name>(<parameter types>) **RETURNS** <return type> **AS**
<code>

- Example:



Languages

- Functions in Postgres can be written in different languages
- We will look at two of them:
  - SQL – you know this!
  - PL/pgSQL – new stuff!
    - Postgres version of Oracles PL/SQL (Procedural Language/SQL)
    - The language is procedural in the sense that (unlike SQL) programs are written as sequences of instructions
    - Similar to general purpose languages (like C, Java) in theory, similar to SQL in syntax

Similar to the table in the assignment

Table: WaitingList

student	course	position
Student1	TDA357	1
Student2	TDA357	2
Student1	TDA143	1

What does this function do?

```
CREATE FUNCTION nextNumber(CHAR(6))
  RETURNS BIGINT AS
$$ SELECT COUNT(*)+1 FROM WaitingList WHERE course=$1
$$ LANGUAGE SQL;
```

- Returns the next position a student should get in WaitingList (assuming the database is currently consistent)
  - nextNumber('TDA357') results in 3
  - nextNumber('TDA143') results in 2
  - nextNumber('XYZ123') results in 1
- Could be a really useful thing!

Creating functions

- First create all the tables, views etc. that the function is using, then run CREATE FUNCTION

```
CREATE Table WaitingList(
   student TEXT,
   course CHAR(6),
   position INT,
   PRIMARY KEY(student,course)
);

CREATE FUNCTION nextNumber(CHAR(6))
   RETURNS BIGINT AS
$$ SELECT COUNT(*)+1 FROM WaitingList WHERE course=$1
$$ LANGUAGE SQL;
```

Functions are part of the design!

Using functions

Table: Courses

code	Name
TDA357	Databases
TDA143	Programming
XYZ123	Fake course

Table: WaitingList

student	course	position
Student1	TDA357	1
Student2	TDA357	2
Student1	TDA143	1

- In SELECTS:

```
SELECT nextNumber('TDA357');
```

```
 nextnumber
------------
          3
```

```
SELECT code, nextNumber(code)
  FROM Courses;
```

```
  code   | nextnumber
---------+------------
 TDA357  |          3
 TDA143  |          2
 XYZ123  |          1
```

- Even more useful for this one, in INSERTs:

```
INSERT INTO WaitingList VALUES
    ('Student1','XYZ123',nextNumber('XYZ123'));
```

If the database is consistent, always gets the right number

Still not completely safe

```
INSERT INTO WaitingList VALUES
    ('Student1','XYZ123',nextNumber('XYZ123'));
```

- This goes a long way to make sure we use the right position, but:
  - The two codes need to match
  - It's still possible to inserts without using the function
  - Deleting from the list creates "holes", and then using the function causes primary key collisions ☹
- Let's try and fix that!

The assertion approach

- Idea: Write a query that finds courses with corrupted waiting lists
- Then write an assertion that the query must give 0 rows
- Observation: A course list is corrupted under any of these conditions:
  - Its lowest position for the course is not 1
  - Its highest position is not equal to its total number of waiting students

```
CREATE ASSERTION No_invalid_positions AS
CHECK NOT EXISTS
  (SELECT course FROM WaitingList
   GROUP BY course
   HAVING MIN(position)!=1 OR MAX(position)!=COUNT(*));
```

- Pros: Looks awesome ☺
- Cons: Assertions don't work in Postgres ☹. Would prevent most deletions ☹

Triggers

- Triggers are procedures (functions) stored on the server, executing when certain actions are taken (like updating, inserting or deleting from a table)
- Postgres syntax example:

```
CREATE FUNCTION <trigger function name>()
  RETURNS trigger AS $$
<Trigger code>
$$ LANGUAGE plpgsql;

CREATE TRIGGER <trigger name>
  AFTER DELETE
  ON <Table name>
  FOR EACH ROW
  EXECUTE PROCEDURE <trigger function name>();
```

A function with a special return type

Uses PL/pgSQL

When is the trigger executed? Possible values:
[BEFORE/AFTER/INSTEAD OF] [DELETE/UPDATE/INSERT]

When are triggers useful?

- When modelling events
  - Something is supposed to happen when a user takes certain actions
  - Like in the assignment: When a student is unregistered from a course, another student from the waiting list may take its place
- Cross-row or cross-table constraints
  - Like a more powerful check constraint, ensure invariants across tables that are more complicated than uniqueness/reference constraints
  - Just like assertions, but we specify when they should be checked

Let's make a simple trigger

- When a student is deleted from the waiting list, "compact" the positions
  - Remove the hole in the consecutive positions for a course created by removing the student from the list
  - We assume the table is consistent before the deletion
- Observation: Can be expressed as a single update, decreasing position by one for all higher positions than the deleted one

The trigger function

- The variable OLD is a special variable for triggers on updates/deletes
- It contains the values of a row that has been (or is about to be) deleted

```
CREATE FUNCTION compact()
  RETURNS trigger AS $$
BEGIN
  UPDATE WaitingList SET position = position-1
    WHERE course = OLD.course AND position > OLD.position;
  RETURN OLD;
END;
$$ LANGUAGE plpgsql;
```

Terminates the function

Course and position values of a deleted row

Creating the trigger itself

- The trigger function is just a function, if it is not called it does nothing
- This code tells Postgres to execute the function AFTER a row has been deleted from WaitingList:

```
CREATE TRIGGER waiting_deleted
   AFTER DELETE
   ON WaitingList
   FOR EACH ROW
   EXECUTE PROCEDURE compact();
```

The function we created on the previous slide

- The trigger needs to be created after creating the function and the table
- Note that if the delete affects multiple rows, the function is called once for each row (not once for the whole DELETE statement!)

Effect of the trigger

Table: WaitingList

student	course	position
Student1	TDA357	1
Student2	TDA357	2
Student1	TDA143	1

```
DELETE FROM WaitingList WHERE student = 'Student1';
```

Table: WaitingList

student	course	position
Student2	TDA357	1

Student2 was bumped to position 1, sweet!

- Note: The trigger was executed twice here
  - For the TDA143 row, it didn't really do anything (0 rows updated)

Triggers and errors

- Updated/deletes that execute triggers are still atomic
  - If there is an error for any row, nothing is changed
- Triggers can raise errors (`RAISE EXCEPTION '<error message>';`)
- Conclusion: We can use Triggers as a kind of 'poor mans assertions', where we write a function that gives an error for invalid operations
  - The problems of assertions are avoided because we have to specify exactly when the condition is checked
- New task: Use this to prevent insertions on incorrect positions

IF-statements and variables in (Postgres) PL/SQL

```
-- Raise an error if the inserted row has a bad position
CREATE FUNCTION valid() RETURNS trigger AS
$$DECLARE cnt INT;
BEGIN
  SELECT COUNT(*) INTO cnt
    FROM WaitingList
    WHERE course = NEW.course;

  IF (NEW.position = cnt) THEN
    RETURN NEW;
  ELSE
    RAISE EXCEPTION 'invalid position';
  END IF;
END;
$$ LANGUAGE plpgsql;
```

Declare an integer variable named cnt

Assign the variable the result of a (single line) query

Use IF/ELSE to either terminate gracefully or raise an error

NEW contains a value that has just been inserted/modified for INSERT/UPDATE triggers

Creating the INSERT-trigger

- The valid function (from previous slide) should be called after each insert (still done before the change is actually visible to anyone)

```
CREATE TRIGGER waiting_inserted
   AFTER INSERT ON WaitingList
   FOR EACH ROW
   EXECUTE PROCEDURE valid();
```

- If the call to valid() raises an error for any inserted row, all changes are "rolled back" to before the INSERT-statement was executed

Variables

- Like local variables in Java and similar languages
- Declared in a single DECLARE block at the start of the code (after $$)
- Each declaration is terminated with ;

```
$$
DECLARE
   cnt INT;
   myBool BOOLEAN;
BEGIN
...
```

Declares two variables

SELECT ... INTO

- Used to run a query and store the result in a variable (declared earlier)
- Either raises an error or gives null/first row if the query does not give exactly one row, depending on DBMS and setting
- Only use things like
  `SELECT credits INTO creds FROM Courses WHERE code=x;`
  if you know x is a valid code
- Tip: Simple aggregates always give one row (possibly containing null)

```
SELECT MAX(credits) INTO creds
  FROM Courses WHERE code=x;
```

  - Variable creds will be null if x is not an existing course code

IF-statements

- You know them, you love them.
- Syntax for if-elseif-else:

```
IF (<condition>) THEN
   ...
ELSIF (<condition> THEN
   ...
ELSE
   ...
END IF;
```

- Both ELSIF and ELSE are optional
- Will run ELSE when condition is UNKNOWN
- In Postgres, you can have queries in the conditions, in original PL/SQL you can not (need to use variables)

Aggregate functions are your friends

- To test something like "Is this student registered for this course", write a query that counts the number of registrations the student has for the course (will be 1 or 0)
- Alternatively, you can use EXISTS

```
IF (EXISTS (SELECT *
            FROM WaitingList
            WHERE course=NEW.course)
   ) THEN …
```

Check if NEW.course has at least one waiting student

Are we safe now?

- Consider the WaitingList(student,course,position) example
- We have made sure that INSERT is only allowed with valid positions
- We have made sure that DELETE will compact to consecutive positions
- Are we 100% sure that waiting lists have consecutive positions?
- No! We can still do UPDATEs on waiting list.
- We could add a trigger ON UPDATE that fixes things?
  - Dangerous: The DELETE trigger will perform an UPDATE, which will trigger the ON UPDATE trigger!
  - If we make an UPDATE trigger that performs UPDATE, we may get an infinite recursive function call ☹

Triggers on views

- Views are amazing for providing a useful interface for applications
- We can select from "tables" that seem to have lots of redundancy, but actually they just reflect data from redundancy-free tables
  - Example: the PassedCourses view that contains credits for each grade (in a proper table this would violate BCNF!)
- Unfortunately, we can SELECT from views, but not INSERT or DELETE
- Triggers changes that!
  - E.g. by writing a trigger INSTEAD OF INSERT ON <view name>, we can do INSERT INTO <view name> VALUES(…) to execute the trigger

INSTEAD OF INSERT on views

- The NEW variable contains the values for an inserted row
  - Uses the columns and types in the view, not in any underlying tables
- Will not automatically insert anything anywhere, the trigger will have to execute INSERT on the underlying tables for anything to happen
  - The row that was added may not show up when selecting from the view
- Should always return NEW unless there is an error

```
CREATE FUNCTION Insert_function() RETURNS trigger AS $$
BEGIN
    -- Code goes here
    RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER Insert_trigger
  INSTEAD OF INSERT ON <view name goes here>
  FOR EACH ROW EXECUTE PROCEDURE Insert_function();
```

INSTEAD OF DELETE on views

- The OLD variable contains a row from the view that matches the WHERE-clause of an executed DELETE-query
- Will report "X rows deleted", which really means the trigger was executed for X rows in the view, the database may not have been changed at all
- Should always return OLD

```
CREATE FUNCTION Delete_function() RETURNS trigger AS $$
BEGIN
   -- Code goes here
   RETURN OLD;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER Delete_trigger
  INSTEAD OF DELETE ON <view name goes here>
  FOR EACH ROW EXECUTE PROCEDURE Delete_function();
```

Should I write triggers on tables or views?

- Writing simple assertion-like triggers on tables is generally OK
- If your triggers do modifications (INSERT/UPDATE etc) to the database, it may be more appropriate to write it on views
- One nice way to think about it: INSERT/UPDATE/DELETE on tables are our internal operations that need to be used very carefully (like private methods)
- Operations on views are our exported interface, and all operations are safe by design, there is no way to corrupt the database by inserting into our views
- When designing the triggers on views we modify the tables directly, but never from applications that use them

Triggers in the assignment

- You will be writing triggers on the Registrations view (the one containing student, course and status (registered/waiting)
  - For DELETE, this means that you can write a single query to unregister a student either from the waiting list or the registration list
- You will use them both to raise errors for invalid modifications and to model events (moving from waiting list to registrations)
- All triggers in the assignment should be written on views!

Debugging triggers

- Stuff not working? Is you IF doing THEN when it should be doing ELSE?
- You can place lines like these anywhere in your trigger to inspect the value of a variable at that position:

```
RAISE NOTICE 'new course: %', NEW.course;
RAISE NOTICE 'variable is: %', cnt;
```

- Text will show up in the output of psql or the 'messages'-tab in pgAdmin
- If no text shows up no matter where you put the code, it means your trigger isn't executed (e.g. because you run a DELETE that matches no rows)
  - Or possibly that you have disabled notices (with client_min_messages)
- CLEAN UP YOUR DEBUG CODE BEFORE SUBMITTING YOUR SOLUTION!

More PL/SQL features

- Assignments: You can do "simple assignments" (no queries):

```
variable := variable * 2 ;
```

- Loops

```
LOOP EXIT WHEN counter = n ;
 counter := counter + 1 ;
END LOOP ;
```

- *Cursors* can be used to loop over the contents of a query
  - Overusing cursors is a common problem for students (e.g. instead of a query that finds all missing prerequisites, they loop through prerequisites and check each with a query)
  - Think relationally when possible, not procedurally!
- All Postgres PL/SQL features: https://www.postgresql.org/docs/15/plpgsql.html

Even more PL/SQL: Recovering from errors

- A code block (BEGIN <code> END;) can have an EXCEPTION clause
- This is like try/catch in Java
- Syntax:

```
BEGIN
    <code that may cause exception>
EXCEPTION
   WHEN <error code>
     <code that handles exception>
END;
```

- There are error codes for violated unique constraints (unique_violation), foreign key violation (foreign_key_violation) and about a million other things here: https://www.postgresql.org/docs/15/errcodes-appendix.html
- You can catch errors you throw yourself, but it's better to not throw them

Side note: Inserting selections

- It's possible to insert the result of selection:
  ```
  INSERT INTO Table2 (SELECT * FROM Table1);
  ```
- Inserts all resulting rows (result types must match columns)
- Can use WHERE, and all other features of queries
- Means INSERT triggers may run multiple times for a single INSERT
- Useful in triggers to copy data from one table to another

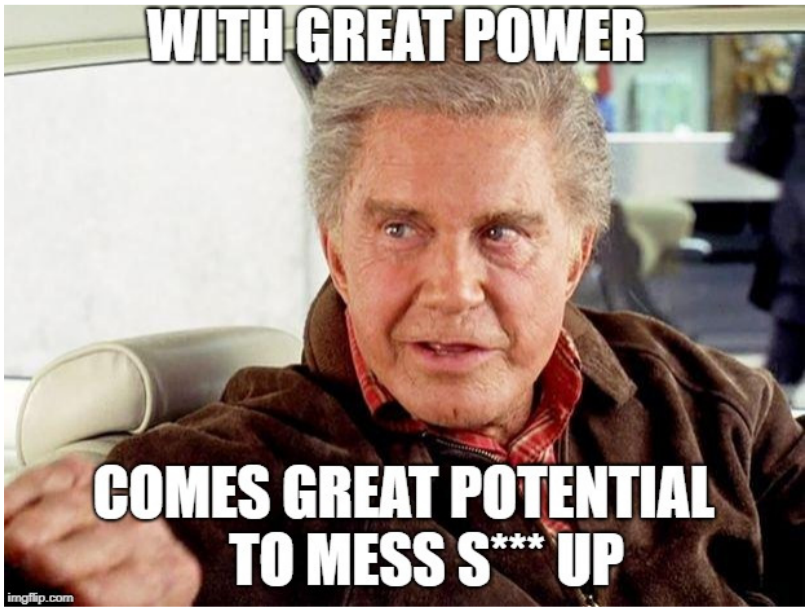
Trigger return values

- Triggers usually return NEW or OLD (New for INSERT/UPDATE and OLD for DELETE)
- Using assignments it's possible to change the values in NEW before returning it. For INSERT/UPDATE triggers, this will modify the inserted/modified row
  - Can be used to make a much nicer version of the insert trigger for WaitingList, that automatically assigns the correct position to the inserted row instead of raising an error
  - Does not work for inserts on views
- If a BEFORE trigger returns NULL, it will skip the update for that row and proceed without any error

Triggers are powerful tools

… but they are also complex to make

- You may introduce bugs in your triggers
- Performance may be problematic with lots of triggers
  - Not an issue if the trigger is executed every time a student registers
  - May be an issue if it is run every time a webpage is refreshed



Avoiding triggers: Using views as constraints?

- Sometimes it is possible to implement a constraint using a view
- If an attribute has a lot of dependencies on other rows/tables, perhaps it should not be a column in a table, but merely in a view?
- Example: A constraint like "nrStudents should be the actual number of registered students for the course"
- In fact, the whole WaitingList example could be implemented by having TIMESTAMPS or relative numbers in the table, and showing the absolute positions only in a View