

---

Introduction to Databases, Fall 2004  
IT University of Copenhagen

**Lecture 8, part I: Constraints and triggers**

October 22, 2004

Lecturer: Rasmus Pagh

---

# — Today's lecture —

---

## Constraints and triggers

- Keys
- Assertion-based constraints (tuple-level, schema-level)
- Foreign keys
- Triggers

## XML for data exchange

- Semistructured data and XML.
- Defining XML formats using XML schemas.

## — **Reminder: Course evaluation** —

- Your feedback is appreciated!
- Ensures quality of studies.
- Influences new study structures (“strukturkommissionen”).

## — What you should remember from previously —

In this lecture I will assume that you remember:

- What a key of a relation is.
- Referential integrity constraints in E/R diagrams.
- How to convert an E/R diagram into relations.
- The SQL used when modifying or adding tuples in a relation.

---

**Next: Keys.**

---

## — Keys in SQL? —

---

So far we used the concept of a **key** of a relation only for normalization.

However, keys play an important role in SQL, because specifying the values of key attributes is a way of referring to a *unique tuple* in a relation.

Since updates (e.g., entered by users of the database) could violate that certain attributes form a key, RDBMSs offer to check this.

## — Declaring a primary key —

---

One key of a relation may be declared as **primary**. This is done by adding to the relation schema a line of the form:

```
PRIMARY KEY (<list of attributes>)
```

If the primary key has just one attribute, we may instead write PRIMARY KEY immediately after the definition of the data type of the attribute, e.g.:

```
id INT PRIMARY KEY,
```

NULL values are not allowed in attributes of a primary key.

## — Declaring other keys —

---

If we want the DBMS to check other key constraints, we may add to the SQL relation schema any number of lines of the form:

```
UNIQUE (<list of attributes in key>)
```

Uniqueness is *not* guaranteed for tuples having NULL values in the key attributes. However, NULL values can be prevented by adding a NOT NULL constraint after the declaration of each key attribute.



## — When a key constraint is violated —

When a key constraint is violated, an error message is produced.

The **state** of the database (i.e., the data it contains) is restored to what it was *before* the action that caused the violation.

Updates in SQL are grouped in units called **transactions** (more about transactions in two weeks).

Constraint-violating transactions are undone (or **rolled back**).

---

**Next: Assertion-based constraints.**

---

## CHECK constraints

---

Another useful kind of constraints, called CHECK constraints, are **assertions** (i.e., conditions that must be true) about attributes or tuples of a relation.

- A CHECK constraint on an attribute is checked every time
  - a value of this attribute is modified.
  - a new tuple is inserted.
- A CHECK constraint on tuples is checked every time
  - an attribute value changes.
  - a new tuple is inserted.
- If a constraint is violated, the current transaction is rolled back, and an error message is produced.

## — Writing attribute-based CHECK constraints —

A constraint  $C$  on an attribute is declared by writing

```
CHECK  $C$ 
```

immediately after the datatype definition.

The condition  $C$  may refer to other attributes of the relation, and even to other relations, using a subquery.

(However, Oracle does not allow subqueries in  $C$ .)

### Examples:

- `percentage INT CHECK (percentage >= 0 AND percentage <= 100)`
- `cpr CHAR(10) CHECK (cpr IN (SELECT cpr FROM students))`

## — Writing tuple-based CHECK constraints —

A constraint  $C$  on tuples is declared by adding the line

```
CHECK  $C$ 
```

to the relation schema definition.

The only difference to attribute-based CHECK constraints is *when* the constraint is checked.

### Examples:

- CHECK (upper-bound => lower-bound)
- CHECK (cpr IN (SELECT cpr FROM students))

## — Schema-level assertions —

---

The SQL standard proposes a general form of assertions, or **general constraints**.

These are declared as part of the database schema, and may assert the truth of any condition  $C$ .

**Semantics:** Any transaction that makes  $C$  false will be rolled back.

Unfortunately, assertions are not implemented in major DBMSs (including Oracle), the reason being that they can be hard to check efficiently.

[Box from G UW p. 339 shown on slide]

---

**Next: Foreign keys.**

---

## Foreign key constraints

---

A **foreign key constraint** on an attribute is a constraint saying that its attribute values can *always be found in exactly one place in another relation*.

Foreign key constraints are typically used to express **referential integrity**, i.e., that values supposed to refer to tuples in other tables indeed do so.

If we want the DBMS to check foreign key constraints, we may add to the SQL relation schema any number of declarations of the form:

```
FOREIGN KEY (<attribute name>)  
REFERENCES <table name>(<attribute name>)
```



## — Composite foreign keys —

---

Foreign keys may be **composite**, i.e., consist of several attributes.

The syntax for declaring composite foreign keys is the obvious extension of what we saw before:

```
FOREIGN KEY (<list of attribute names>)  
    REFERENCES <table name>(<list of attribute names>)
```

## — Semantics of a foreign key constraint —

Suppose the schema for relation  $R$  contains the declaration

FOREIGN KEY  $(A_1, \dots, A_n)$  REFERENCES  $S(B_1, \dots, B_n)$ .

Then the schema for relation  $S$  *must* contain a declaration like

UNIQUE  $(B_1, \dots, B_n)$ .

This means that the DBMS checks:

- That every tuple in  $\pi_{A_1, \dots, A_n}(R)$  that has no NULL value is also in  $\pi_{B_1, \dots, B_n}(S)$ .
- The UNIQUE constraint on  $B_1, \dots, B_n$

## — Problem session (5 minutes) —

---

What is the difference (if any) between the CHECK constraint

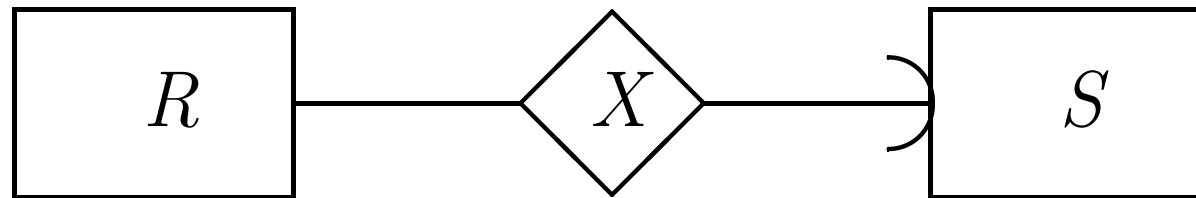
```
cpr CHAR(10) CHECK (cpr IN (SELECT cpr FROM students))
```

and the referential integrity constraint

```
cpr CHAR(10) REFERENCES students(cpr)
```

## — Referential integrity from E/R diagrams —

Suppose we based our relation schema on an E/R diagram, where there was a referential integrity constraint on a relationship  $X$ .



If the key for entity set  $S$  is  $\{A_1, \dots, A_n\}$ , then

- The relation corresponding to  $R$  has  $A_1, \dots, A_n$  as attributes (it was combined with the relation for  $X$ , as described in GUW 3.2.3).
- The relation corresponding to  $S$  has  $A_1, \dots, A_n$  as primary key.

We can express the referential integrity constraint in the schema for  $R$ :

FOREIGN KEY ( $A_1, \dots, A_n$ ) REFERENCES S( $A_1, \dots, A_n$ ).

## — Maintaining referential integrity —

The default (i.e., standard) policy when a transaction violates a foreign key constraint is to roll the transaction back.

However, for each referential constraint we may choose from two other policies for handling changes *to the referenced relation*:

- **The cascade policy:**

- If the foreign key attribute values of a tuple were changed, change all references to this tuple to the new value.
- If a tuple is deleted, delete all tuples referencing it.

- **The set-null policy:**

- If some reference became invalid, set all its attribute values to NULL.

## — Problem session (5-10 minutes) —

Consider the E/R diagram of G UW Figure 2.18 (with suitable attributes added). You should convert the diagram into a suitable relational database schema, including:

- Name and attributes of each relation.
- Referential integrity constraints.

Suppose we delete a studio from the database. What would happen to other relations if we use:

- The cascade policy.
- The set-null policy.

---

**Next: Triggers.**

---

# Triggers

---

**Triggers** (or **event-condition-action rules**) is a general mechanism for:

- Enforcing constraints, and more generally
- Making the DBMS perform actions on certain events.

The definition of a trigger consist of an event, a condition, and an action.

- Triggers are awakened (or triggered) when the **event**, a certain change to the database, occurs.
- If the **condition** associated with the trigger is true, then the **action** is performed.

[Figure 7.8 shown on slide]



# — Triggers in SQL —

---

Key features of triggers in SQL:

- Triggering events are insertions, deletions, and updates of tuples.
- The action can be any SQL statement.  
(But most RDBMSs have restrictions on the SQL allowed in the action.)
- The action can refer to values from both before *and* after the event.
- The action can be performed either
  - After each event that activates the trigger, or
  - At the end of each transaction where one or more events activated the trigger.

## — Trigger definition syntax, simplified —

Syntax in Oracle (differs slightly from SQL definition):

```
CREATE TRIGGER <name of trigger> AFTER
INSERT | DELETE | UPDATE
    [OF <attribute name>] ON <name of relation or view>
[REFERENCING OLD AS <name>, NEW AS <name>]
[FOR EACH ROW
    [WHEN <condition>]]
BEGIN
    <SQL commands>
END;
```

Vertical lines | between alternatives. Brackets [] around optional parts.

Variables in <SQL commands> must be prefixed by semicolon (:old.a).

## — Instead-of triggers and views —

---

A special kind of triggers, **instead-of triggers**, supported by e.g. Oracle, can be used to specify what should happen when a view is updated.

- This makes it possible to update some views that are not updateable.
- If the view is updatable, the action of the trigger is executed *instead of* the update itself.

## — Most important points in this lecture —

As a minimum, you should after this week:

- Know how to declare key constraints and referential integrity (i.e., foreign key) constraints in SQL.
- Understand the basic mechanisms for maintaining referential integrity.
- Know how to declare tuple-based CHECK constraints, and know how these are checked.
- Understand how to define triggers, and the mechanism for executing triggers in SQL.