Databases - Constraints and Triggers
This lecture
describes how to enforce constraints and execute triggers.
SQL allows one to create “active” elements in a database.

An active element in an expression or statement that is stored in the database and executed at the appropriate “time”.

The time may be when an insertion, update or deletion on a relation is attempted, or equally if changes result in a boolean value being TRUE.

We have discussed some active elements already. Key constraints (PRIMARY/FOREIGN) are examples of statements that have effect when changes are made to the database.
Why and where

Updates to databases can be wrong in a variety of ways. For example,

- There may be typographical and transcription errors
- Data values may be outside of an allowed range
- Data values have an interrelationship

These constraints can be written at the application level, however it is better to store these checks in the database and let the DBMS administer them. Duplication is avoided and the check is never “forgotten”.

Support for integrity constraints is present in most DBMSs, but support for “checks”, “assertions” and “triggers” is considerably weaker.
Primary Keys

An attribute or set of attributes of a relation that are a key are identified with the keywords `PRIMARY KEY` or `UNIQUE`.

Collectively the value(s) of the attributes of the key are unique, and thereby uniquely identify the tuple (row) of the relation (table).

Any update that effects the values of the key, or violate the key integrity will result in an error state for the DBMS.
Foreign keys

Referential-integrity constraints can be declared with the key word FOREIGN KEY. This constraint asserts that the value appearing in one relation must also appear as the primary-key component(s) of another relation.

Consider the following schema

```sql
CREATE TABLE S (  
  ps INT  
  ...  
  FOREIGN KEY (ps) REFERENCES M(pm)  
);
```
Foreign key exceptions

A runtime exception to referential-integrity will occur when we

- insert an $S$ tuple, with a $ps$ that is a not NULL value and is not the $pm$ component of any $M$ tuple
- update an $S$ tuple to change the $ps$ to a nonNULL value that is not the $pm$ component of any $M$ tuple
- delete an $M$ tuple and its $pm$ component, which appears as the $ps$ component of some $S$ tuple
- update an $M$ tuple that changes a $pm$ component, which is the same as the $ps$ component of some $S$ tuple
Not-Null constraints

A simple attribute constraint is NOT NULL. The effect is to disallow tuples in which the particular attribute is NULL.

This use of NOT NULL can be important for UNIQUE attributes. In the case of UNIQUE attributes, NULL values are “accepted and ignored”.

This means NULL is considered a legitimate value AND multiple instances of NULL values for an attribute is not considered a violation of uniqueness.

This can be avoided by declaring the attribute to be NOT NULL.
**CHECK constraints - attribute**

More complex constraints can be attached to an attribute by use of the keyword **CHECK** followed by a conditional expression which must hold for every value of that attribute.

Such **CHECKs** are likely to be limits on values, arithmetic inequalities, or restrictions to an enumeration of values (**ENUM** also achieves this).

The **CHECK** is executed whenever a tuple is assigned a new value for the attribute to which the **CHECK** is attached.

The constraint is checked only when changes occur to the attribute to which the constraint is associated. It is possible for this constraint to be violated if other values involved in the constraint are changed.
Example attribute-based **CHECK** constraints

```plaintext
p INT CHECK( p >= 0 AND p <= 100)
```

Require the value of `p` is an integer between 0 and 100 inclusive.

```plaintext
gender CHAR(1) CHECK( gender IN ('F', 'M'))
```

Has the same effect as an ENUM definition.

```plaintext
ps INT CHECK( ps IN (SELECT pm FROM M))
```

An *erroneous* simulation of referential-integrity maintenance.
**CHECK constraints - tuple**

If we declare a `CHECK` constraint in a schema at the same level as an attribute declaration, then that is interpreted as a condition on the tuple of the relation. The attributes of the relation may be referred to in the expression.

```sql
CREATE TABLE M (  
title CHAR(5),
...,  
gender CHAR(1),
dob DATE,
CHECK(gender = 'F' OR title NOT LIKE 'Ms.%')
);
```

MySQL parses but ignores `CHECK` constraint declarations.
More powerful *active* elements

The previous *active* elements were associated with tuples or components of tuples. More powerful elements are associated with neither, and are on a par with tables. These are *assertions* and *triggers*.

An assertion is a boolean-valued SQL expression that is always true.

A trigger is a series of actions associated with certain events.

Assertions are easy to define, very difficult to implement. The DBMS must determine if modifications affect an assertion’s truth - *declarative*.

Triggers specifically identify what the DBMS needs to do - *procedural*.
Asserts and Triggers

Assertions

CREATE ASSERTION <a-name> CHECK (<condition>)

The condition must be true when the assertion is created and must remain true or the database modification is rejected.

CREATE ASSERTION TooRich CHECK (
    (NOT EXISTS
    (SELECT company-name
    FROM Company, Executives
    WHERE ceo = eid AND networth < 10000000
    )
    );

Ensure that no company has as its CEO someone whose net worth is less than $10,000,000.

MySQL does not implement assertions.
Triggers are also known as event-condition-action or ECA rules.

1. Triggers are awakened by certain programmer specified events, eg insert, delete or update on a relation or on a transaction end.

2. Triggers test a condition. If the condition does not hold, nothing else is done.

3. If the condition is satisfied, the action associated with the trigger is performed. It can modify the effects of the event, or even abort the transaction. It can perform any sequence of DB operations that may have nothing to do with the particular event.
SQL Triggers

Features of an SQL trigger are;

1. The check of the *condition* and the *action* may be executed on the state of the database *before* the triggering event is executed, or the state that exists *after* the event is executed.
2. The condition an action can refer to the *old* and the *new* values of tuples after an update.
3. It is possible to limit update events to a certain attribute or set of attributes.
4. A trigger may execute once for each modified tuple (*row-level trigger*) or once for all tuples that are changed by an SQL statement (*statement-level trigger*).
**Example Trigger**

CREATE TRIGGER NetWorthTrigger
AFTER UPDATE of netWorth on MovieExec
REFERENCING
OLD ROW as OldTuple,
NEW ROW as NewTuple
FOR EACH ROW
WHEN (OldTuple.netWorth > NewTuple.netWorth)
UPDATE MovieExec
SET netWorth = OldTuple.netWorth
WHERE cert = NewTuple.cert;

This trigger is designed to foil (by undoing) any update on an attribute (netWorth) with a lower value.

There is limited support for triggers in MySQL, but restricted to users with SUPER privileges. As of Version 5.1.6 triggers can be created and dropped when appropriate privileges are set.