Deletion

```
DELETE FROM R WHERE <condition>
```

Removes every tuple satisfying the condition from R

```
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
      movieYear = 1942 AND
      starName = 'Sydney Greenstreet';
```

All three conditions may not be neccessary to identify a tuple (e.g., just specify key):

```
DELETE FROM StarsIn
WHERE movieTitle = 'The Maltese Falcon' AND
      movieYear = 1942;
```

Deleting several tuples at the same time

```
DELETE FROM MovieExec
WHERE netWorth < 10000000
```

Removes all tuples for which the value of `netWorth` is less than 10.000.000

Updates

Update: Modifies part of a tuple

```
UPDATE MovieExec
SET name = 'Pres.' || name
WHERE cert# IN
    (SELECT presC#
     FROM Studio);
```

name on left of SET clause is the new value; name on right is the old one

Tables

Three types

- Stored relations or tables
- Views. Created when needed. To be studied later
- Temporary tables (e.g., `WITH` clause)

Data Types

- Character strings of fixed or varying length
  - `CHAR(n)`: fixed-length string of up to n characters
  - `VARCHAR(n)`: string of up to n characters
- Bit strings
  - `BIT(n)`: length n
  - `BIT VARYING(n)`: up to length n
- `BOOLEAN`: Boolean strings. Values can be `TRUE`, `FALSE`, `UNKNOWN` (to be explained later)
- `INT` or `INTEGER`: integers (also `SHORTINT`)

Types, continued

- Floating point
  - `FLOAT`, `REAL`
  - `DOUBLE PRECISION`
  - `DECIMAL(n,d)`. For example 0123.45 is of type `DECIMAL(6,2)`

`DATE` and `TIME`. Special form of strings:

  - `DATE ’1948-05-14’`
  - `TIME ’15:00:02.5’`

Table declarations

```
CREATE TABLE Movies (
   title      CHAR(IOO),
   year       INT,
   length     INT,
   genre      CHAR(10),
   studioName CHAR(30),
   producerC# INT
);

CREATE TABLE MovieStar (
   name      CHAR (30),
   address   VARCHAR(255),
   gender    CHAR(1),
   birthdate DATE
);
```

Modifying tables

- `DROP TABLE R;`

  Deletes table *and all of its tuples*

- Modify table. Syntax:
  - `ALTER TABLE`
  - Name of relation
  - Options. Two are:
    - `ADD` followed by attribute name (and type)
    - `DROP` followed by attribute name

Example

- `ALTER TABLE MovieStar ADD phone CHAR(16);`
- `MovieStar` now has five attributes
- What happens to the tuples?
- Normally, the value will be set to `NULL` (but one can specify a different default)
- Another example:

  `ALTER TABLE MovieStar DROP birthdate`

  deletes the birthdate attribute (the values are lost)

Default values

- Default value (instead of `NULL`)
- `ALTER TABLE MovieStar ADD phone CHAR(16) DEFAULT ’unlisted’;`
- Other examples

```
gender CHAR(1) DEFAULT ’?’

birthdate DATE DEFAULT DATE ’0000-00-00’
```

Keys

Two ways to declare:

- Part of attribute declaration (single-attribute keys only)
- Separate statement

Two types of declarations:

- `PRIMARY KEY`
- `UNIQUE`

Semantics: No two different tuples can have the same values for the key attribute(s)

PRIMARY KEY vs. UNIQUE

- Every relation must have exactly one PRIMARY KEY
- A relation may have one or more UNIQUE statements
- PRIMARY KEY attributes cannot be NULL
- UNIQUE attributes may be. Two different tuples may have NULL values for UNIQUE attributes
- An implementation usually creates an efficient access method for PRIMARY KEY attributes

Examples

```
CREATE TABLE MovieStar (
   name      CHAR(30) PRIMARY KEY,
   address   VARCHAR(255),
   gender    CHAR(1),
   birthdate DATE
);
```

```
CREATE TABLE MovieStar (
   name      CHAR(30),
   address   VARCHAR(255),
   gender    CHAR(1),
   birthdate DATE,
   PRIMARY KEY (name)
);
```

Two-attribute keys

```
CREATE TABLE Movies (
   title      CHAR(100),
   year       INT,
   length     INT,
   genre      CHAR(10),
   studioName CHAR(30),
   producerC# INT,
   PRIMARY KEY (title, year)
);
```

Constraints

- Constraints are restrictions on the contents of the database
- The system guarantees their validity
- Attempts to violate them will be rejected by the system
- Two types of constraints: Constraints on a single relation (checked only when modifying this relation) and constraints on the whole database (checked on any modification)
- Examples we have seen: `PRIMARY KEY` and `UNIQUE`

Example of `PRIMARY KEY` and `UNIQUE` constraints

```
CREATE TABLE Student (
    name         CHAR(30),
    codFiscale   CHAR(16),
    matricola    INT,
    PRIMARY KEY (codFiscale),
    UNIQUE (matricola)
);
```

- Also allowed: “`matricola INT UNIQUE`” provided `UNIQUE` refers to a single attribute
- PRIMARY KEY and UNIQUE are quite similar
- One (and only one) PRIMARY KEY is required for every relation
- Any number of UNIQUE constraints are allowed
- PRIMARY KEY is usually used by the system to construct an efficient access mechanism
- PRIMARY KEY attribute(s) can never be `NULL`; UNIQUE attributes can be

Foreign Keys: Second most important class of constraint

```
Studio(name,address, presC#)
MovieExec(name,address,cert#,netWorth)
```

presC# can be seen as a reference to a tuple in MovieExec if

- cert# is declared as a key or as UNIQUE
- For every value of precC# in Studio, there is a tuple in MovieExec with this value for cert#
- Exception: when presC# is NULL

Two formats

```
CREATE TABLE Studio(
    name     CHAR(30) PRIMARY KEY,
    address  VARCHAR(255),
    presC#   INT REFERENCES MovieExec(cert#)
)
```

```
CREATE TABLE Studio(
    name     CHAR(30) PRIMARY KEY,
    address  VARCHAR(255),
    presC#   INT,
    FOREIGN KEY (presC#) REFERENCES MovieExec(cert#)
)
```

Foreign keys can also contain more attributes; in this case only the second syntax is allowed

`FOREIGN KEY` syntax does not require that `cert#` be a key, but it is always used in this way

Modifications

What happens when we try to modify one of the two relations?

Modifying `Studio`

- Deletion: Nothing can go wrong
- Insertion: If we insert a tuple with non-`NULL` `presC#` field which is not present in `MovieExec`, we must reject the insertion
- Same with modification of the `presC#` of an existing tuple. All other updates (i.e., modifications of other attributes) are OK

Modifying `MovieExec`

- Inserting tuples: No problem
- Deleting a tuple whose `cert#` exists in `Studio`: A problem
- Same problem with modifying a `cert#` field that exists in `Studio`

Policies

In the case of changes to `MovieExec`, we have several choices

- Reject the modification
- Cascade
  - In the case of deletion, delete all the tuples in `Studio` that refer to this tuple. There might be tuples in another relation that refer to these, in that case delete them, and so on
  - In the case of update, change the values of `cert#` in `Studio` that refers to this tuple to the new value
- Change the `cert#` fields that refer to the modified tuple to `NULL`

SQL syntax

```
 CREATE TABLE Studio(
    name     CHAR(30) PRIMARY KEY,
    address  VARCHAR(255),
    presC#   INT REFERENCES MovieExec(cert#)
       ON DELETE SET NULL
       ON UPDATE CASCADE
)
```

Constraints on a single relation

Single relation constraints can be on a single attribute or a single tuple

One example: `NOT NULL` says that the attribute is not allowed to be equal to `NULL`. For example

```
presC#  INT REFERENCES MovieExec(cert#) NOT NULL
```

Note that in this case `ON DELETE SET NULL` cannot be used

CHECK constraints

Two types:

- conditions on individual attributes
- conditions on individual tuples

Both are part of table declaration. First type *can* be part of attribute declaration

- In Studio

```
presC#  INT REFERENCES MovieExec(cert#)
            CHECK (presC# >= 1000000),
```

- In MovieStar

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

Syntax

- The condition in CHECK is a SQL condition with BOOLEAN result
- Condition can use other relations, but this is usually a bad idea
- Attempt to define foreign key using CHECK

```
presC# INT CHECK
   (presC# IN (SELECT cert# FROM MovieExec)),
```

- What is wrong with this?

When constraints are checked

- `CHECK` are tested when the relation in which they are declared is changed
- Therefore, this condition is not tested when `MovieExec` is changed
- Foreign keys are tested when either relation changes

Tuple-based constraints: constraints on whole tuple

```
CHECK (gender = 'F' OR name NOT LIKE 'Ms. %')
```

Syntax for single-attribute conditions

Two possibilities

```
CREATE TABLE MovieStar(
    name        CHAR(30)  PRIMARY KEY,
    address     VARCHAR (255),
    gender      CHAR (1) CHECK (gender in ('F', 'M'),
    birthdate   DATE
)
```

or

```
CREATE TABLE MovieStar(
    name        CHAR(30)  PRIMARY KEY,
    address     VARCHAR (255),
    gender      CHAR (1),
    birthdate   DATE,
    CHECK (gender in ('F', 'M')
)
```

First is better, as it only needs to be checked when that attribute is changed

Modfiying constraints

- In `Studio`

  ```
  presC#  INT CHECK (presC# >= 1000000),
  ```

- We want to change the constraint. This can only be done is the constraint is declared with a name

  ```
  presC# INT CONSTRAINT SixDigits
          CHECK (presC# >= 1000000)
  ```

- We can the write

  ```
  ALTER TABLE Studio DROP CONSTRAINT SixDigits

  ALTER TABLE Studio ADD CONSTRAINT SevenDigits
                      CHECK (presC# >= 10000000)
  ```

Modification of constraints

- To modify a constraint it must be given a name when declared
- Examples

  ```
  name CHAR(30) CONSTRAINT NameIsKey PRIMARY KEY
  gender CHAR(1) CONSTRAINT NoAndro CHECK (gender IN (F>, >M)),
  ```

- Delete constraint

  ```
  ALTER TABLE MovieStar DROP CONSTRAINT NameIsKey;
  ```

- Add constraint

  ```
  ALTER TABLE MovieStar ADD CONSTRAINT NameIsKey PRIMARY KEY (name);
  ```

Assertions

- Outside the declarations of tables
- Boolean queries, that must always be true
- Should be checked after every update
- Example

  No studio can have a president with netWorth less than 10.000.000
  (but other executives could)

```
CREATE ASSERTION RichPres CHECK
   (NOT EXISTS
      (SELECT Studio.name
       FROM Studio, MovieExec
       WHERE presC# = cert# AND netWorth < 10000000);
```

Another example

Aggregation

```
Student(matricola, name, course)
```

No course can have more than 50 students

```
CREATE ASSERTION numeroChiusa CHECK
   (NOT EXISTS
     (SELECT course
      FROM student
      GROUP BY course
      HAVING COUNT(matricola) > 50))
```

Triggers

- Triggers are commands that are executed when something “happens” to the database
- We will look at triggers on updates
- Trigger has a condition that is checked when the database is changed
- If the condition holds, the trigger is activated

Example

- Trigger that does not allow reducing the value of `netWorth`
- Why can't we use a constraint?
- Intuition: After a "bad" update, restore the old value

```
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#;
```

```
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#;
```

- Declare trigger and give it a name
- When to apply trigger. Options:
  - `AFTER`, `BEFORE`, `INSTEAD OF`
  - `INSERT`, `DELETE`, `UPDATE`
  - `OF`: Only makes sense (and is optional) for `UPDATE`
  - `ON`: relation name

Accessing data

```
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#;
```

- `REFERENCING` followed by variable names for `OLD` (before modification) and `NEW` (after)
- `OLD` only allowed for update and delete; `NEW` for update and insert
- Options: `ROW` (tuple) and `TABLE` (entire relation)

```
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#;
```

- `FOR EACH ROW` or `FOR EACH STATEMENT` (update statement)
- `WHEN` optional
- SQL statement, or list of SQL statements (with `BEGIN` and `END`)

Another example

- Constraint on `Movie Exec`: Don't allow average of `netWorth` to be less that 500.000
- Cannot do this by checking each tuple
- Needs to be done for the complete update statement
- This example could also be done by an assertion. With a more complicated trigger, we could also say how to fix the update
- We describe a trigger for updates. Similar triggers must also be written for insertion and deletion

```
CREATE TRIGGER AvgNetWorthTrigger
AFTER UPDATE OF netWorth ON MovieExec
REFERENCING
   OLD TABLE AS OldStuff,
   NEW TABLE AS NewStuff
FOR EACH STATEMENT
WHEN (500000 > (SELECT AVG(netWorth) FROM MovieExec))
BEGIN
   DELETE FROM MovieExec
   WHERE (name, address, cert#, netWorth) IN NewStuff;
   INSERT INTO MovieExec
      (SELECT * FROM OldStuff);
END;
```

BEFORE triggers

- Insert tuples into Movies(title,year,length,genre,studioName,producerC#)
- What if we don't know the year?
- Cannot insert a NULL value, since year is part of the key
- Check for NULL and change the value if needed
- We use 1915, which could be done with default values, but more complicated rules can be implemented with triggers

```
CREATE TRIGGER FixYearTrigger
BEFORE INSERT ON Movies
REFERENCING
   NEW ROW AS NewRow
   NEW TABLE AS NewStuff
FOR EACH ROW
   WHEN NewRow.year IS NULL
   UPDATE NewStuff SET year = 1915;
```

Views

- View: Defined by a query
- Creates a virtual table: Can be used by queries *as though* it was another relation
- But the relation is not physically constructed

Used for

- Convenience
- Security: Allow users only to access views, which hide the rest of the data

Why not create physical relations?

- Save space
- No need to update them

Example

```
CREATE VIEW ParamountMovies AS
   SELECT title, year
   FROM Movies
   WHERE studioName = 'Paramount';
```

User may be permitted access only to this relation, hiding many tuples as well as some attributes

Another example:

```
CREATE VIEW MovieProd
  SELECT title, name
  FROM Movies, MovieExec
  WHERE producerC# = cert#;
```

Saves the user from writing a join to find the producer of a movie

Use of Views

Views can be used *as though* they were normal relations

Find movies made by Paramount in 1979

```
SELECT title
FROM ParamountMovies
WHERE year = 1979;
```

One can use relations and views in the same query.

Find all the names of Stars in Paramount movies

```
SELECT starName
FROM  ParamountMovies, StarsIn
WHERE title = movieTitle AND year = movieYear;
```

This is equivalent to

```
SELECT starName
FROM  (SELECT title, year
       FROM Movies
       WHERE StudioName = ’Paramount’
      ) Pm, StarsIn
WHERE Pm.title = movieTitle AND Pm.year = movieYear;
```

Another example: Rename attributes in View

```
CREATE VIEW MovieProd(moveTitle, prodName) AS
   SELECT title, name
   FROM Movies, MovieExec
   WHERE producerC# = cert#;
```

`VIEW` and `WITH`

- Views are always available, until they are deleted
- `WITH` creates a relation than is only available during execution of a query