Defining SQL Tables, Integrity Constraints and Views

CS-A1153: Databases

Prof. Nitin Sawhney

Aalto University
School of Science

Acknowledgements

These slides are based in part on presentation materials created by **Kerttu Pollari-Malmi** and **Juha Puustjärvi** in previous years and on the course text book: **A First Course in Database Systems**, Third Edition. Pearson by Jeffery D. Ullman and Jennifer Widom.

Thanks to **Etna Lindy** & **Ville Vuorenmaa** for translating prior lecture slides for this course.

Learning Goals

- Defining tables (relations) with SQL
- Modifying tables and their contents with SQL
- Defining integrity constraints with SQL:
  - Declaring attribute(s) of one table to be a *foreign key*, referencing attribute(s) of another table.
  - What the database management system will do, if integrity constraints are violated?
- Understanding views, and how to define and use them.

Defining Relations in SQL

Three types of relations can be represented in SQL:

1. **Tables:** Relations stored in the database that can be queried and modified by changing its tuples.

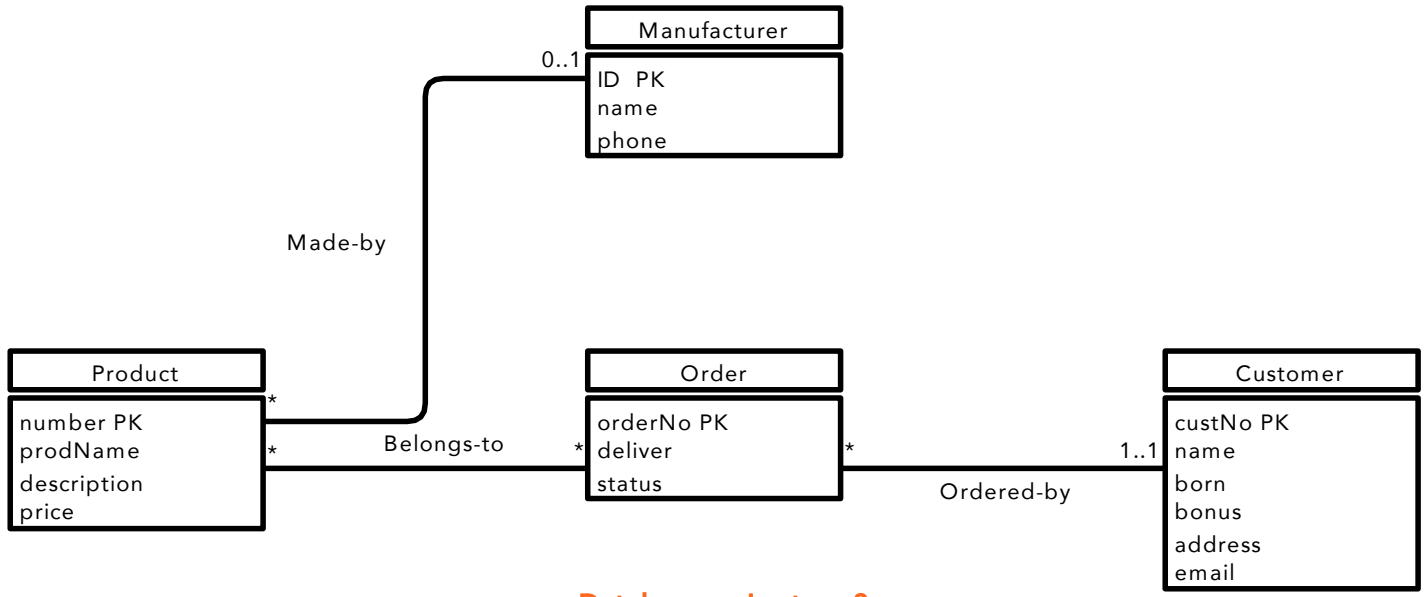
2. **Views:** Relations defined by a computation; not stored but constructed, in whole or in part, when needed.

3. **Temporary Tables:** Relations constructed by the SQL language processor when executing queries or data modifications. These are thrown away, when not needed, rather than stored in the database. *Often for holding temporary results of queries in an operation.*

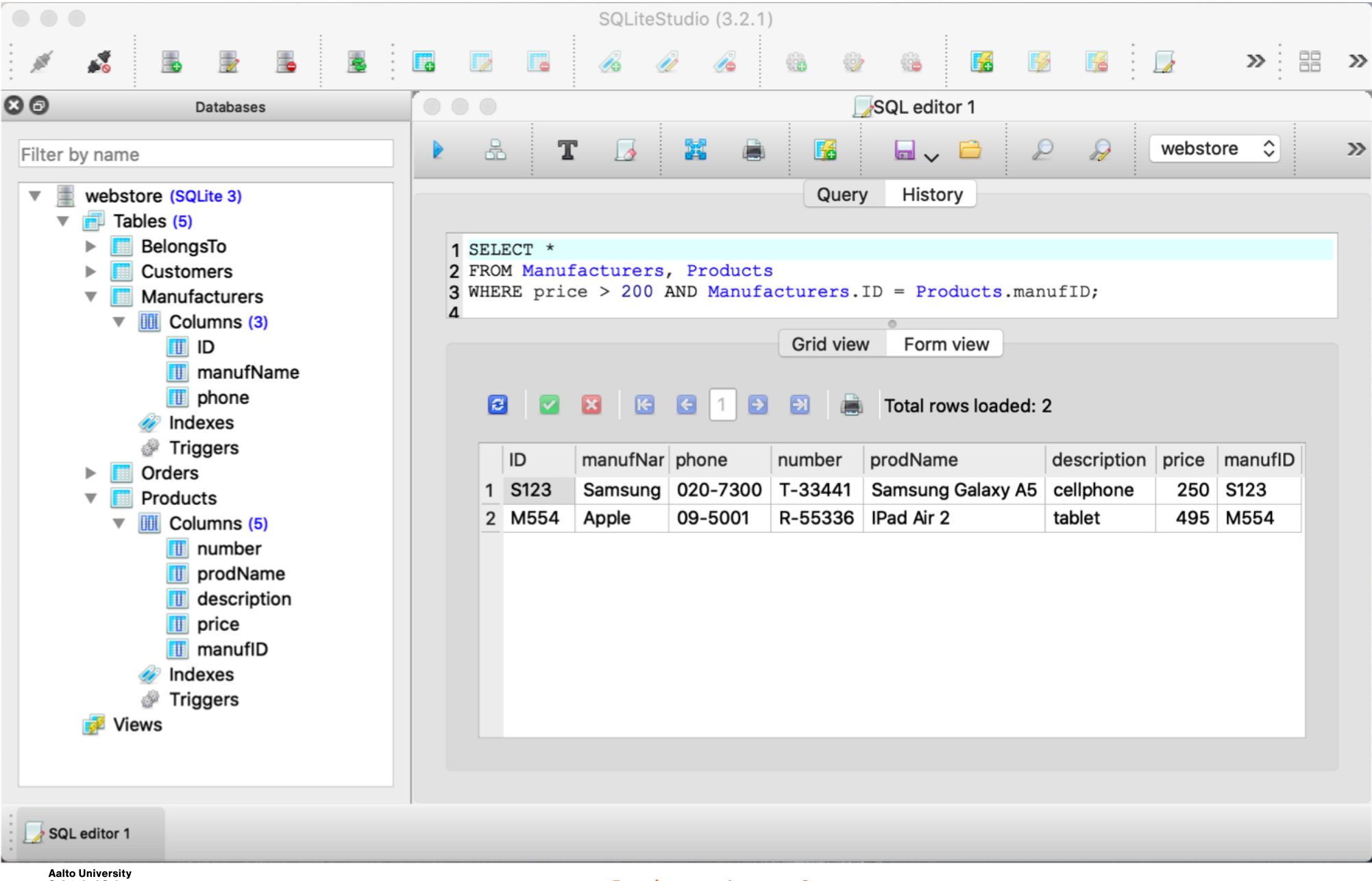
Example Database

- Examples in this lecture use the database from prior lectures consisting of the following relations/tables:

  Customers(custNo, name, born, bonus, address, email)
  Products(number, prodName, description, price, manufID)
  Manufacturers(ID, manufName, phone)
  Orders(orderNo, deliver, status, custNo)
  BelongsTo(orderNo, productNo, count)



Trying Examples using SQLiteStudio



Defining Tables in SQL

- The SQL CREATE TABLE statement declares the schema for a stored relation or table. It specifies a name for the table, its attributes and their data types.

- Example using SQL standard types:

```
CREATE TABLE Customers (
        custNo CHAR(10),
        name CHAR(100),
        born INT,
        bonus INT,
        address CHAR(100),
        email CHAR(100)
);
```

- Later we consider other features of CREATE TABLE for defining key(s) and declaring many forms of constraints and indexes.

Defining Tables with SQLite

- With SQLite data types, the same table would be defined as follows:

```
CREATE TABLE Customers (
        custNo TEXT,
        name TEXT,
        born INTEGER,
        bonus INTEGER,
        address TEXT,
        email TEXT
);
```

Data Types in SQL

1. **Character Strings**:
   CHAR(n): fixed-length string (of n characters)
   VARCHAR(n): variable-length string (of up to n characters)

A string is padded by trailing blanks if it's an attribute of a fixed-length string. For example, 'foo' would be 'foo ' for string type CHAR(5).

2. **Bit Strings**:
   BIT(n) vs. BIT VARYING(n) – bits with fixed or varying lengths

3. **Logical Values:**
   BOOLEAN with attributes TRUE, FALSE and UNKNOWN

Data Types in SQL

4. **Integer Values:**
   INT or INTEGER *(synonymous)*
   SHORTINT *(same as INT, but number of bits permitted may be less)*

5. **Floating-Point Numbers:**
   FLOAT or REAL *(synonymous)*
   DOUBLE PRECISION *(higher precision than REAL)*
   DECIMAL (n,d): represents real numbers with a fixed decimal point
   For example, 0123.45 is a value of type DECIMAL (6,2)
   NUMERIC *(synonymous with DECIMAL, based on implementation)*

6. **Dates and Times:**
   DATE and TIME *(keyword followed by quoted string in special form)*
   For example, DATE '1948–05–14' is 14th of May 1948
   and TIME '15:00:02.5' is two and half seconds past 3 pm.
   Can be compared using regular string comparison operators.

Example in Standard SQL

- Let's define the relation:

  Products(number, prodName, description, price, manufID)

  as a table in standard SQL.
- Possible definition:

```
CREATE TABLE Products(
        number CHAR(10),
        prodName CHAR(80),
        description VARCHAR(200),
        price REAL,
        manufID CHAR(10)
);
```

Data Types in SQLite

- The possible date types for SQL vary based on the database management system.
- In SQLite there are only 4 data types:
  - INTEGER for integers
  - REAL for decimal numbers
  - TEXT for strings
  - BLOB for binary data saved as such
- In SQLite, the Boolean values TRUE and FALSE are represented with integers 1 and 0.
- SQLite uses dynamic typing and does not check if attributes have values of defined types.
- However, SQLite accepts standard definitions.
- Learn more about SQLite data types: http://www.sqlite.org/datatype3.html

Previous Example in SQLite

- Defining the relation:

  Products(number, prodName, description, price, manufID)

  as a table in SQLite.
- Possible definition:

```
CREATE TABLE Products(
        number TEXT,
        prodName TEXT,
        description TEXT,
        price REAL,
        manufID TEXT
);
```

Dates in SQLite

- In SQLite dates can be expressed as strings.
  For example:
  '2020-07-01' means July 1st 2020 and
  '2020-07-01 10:52' means July 1st 2020 at 10:52
- Examine other possibilities:
  http://www.sqlite.org/lang_datefunc.html
- Dates can be compared with comparison operators, for example

```
SELECT *
FROM MovieStar
WHERE birthdate >= '1970-01-01';
```

Note: *The separator in strings (e.g. between year and month) is a regular hyphen. If you copy-paste dates from this pdf file, the separator might be the wrong character.*

Modifying Database Schemas in SQL

- Possible operations:
  - Creating a new table to the database
  - Removing tables from the database
  - Adding new attributes to the table
  - Removing some attributes from the table

Creating a new table was covered in previous slides. Next, we'll look performing other operations with SQL.

Modifying Database Schemas in SQL

- Tables can be removed using DROP TABLE.
  For example, **DROP TABLE** Customers;
- Attributes (columns) can be added to and removed from an existing table using ALTER TABLE.

  For example,
  ALTER TABLE Customers **ADD** phone CHAR(16);

  Adds the attribute phone (with a fixed-length string of 16 characters) to the table Customers.

  Whereas **ALTER TABLE** Customers **DROP** born;

  Removes the attribute born from the table Customers.
  (the latter does not work in SQLite)

Defining Default Values for Tables in SQL

- When defining new tables and adding attributes to old ones, tables can be assigned default values for attributes.
- If any tuple of the table has no other value given for the attribute, then the default value is used.
- The default value can be either NULL or some constant.

Defining Default Values, Examples in SQL

- Example of assigning default values when defining the table in SQL (default values given to bonus and address):

```
CREATE TABLE Customers (
        custNo CHAR(10),
        name CHAR(100),
        born INT,
        bonus INT DEFAULT 0,
        address CHAR(100) DEFAULT '?',
        email CHAR(100)
);
```

- Example of defining a default value when adding an attribute to the table:

```
ALTER TABLE Customers ADD phone CHAR(16) DEFAULT 'unknown'
```

Defining Default Values, Examples in SQLite

- Same example of assigning default values when defining a table in SQLite:

```
CREATE TABLE Customers (
        custNo TEXT,
        name TEXT,
        born INTEGER,
        bonus INTEGER DEFAULT 0,
        address TEXT DEFAULT '?',
        email TEXT
);
```

- Defining a default value when adding an attribute to the table:

```
ALTER TABLE Customers ADD phone TEXT DEFAULT 'unknown'
```

Defining Keys in SQL

- In SQL, an attribute or a set of attributes can be defined as a key to a table in two ways:
  1. The attribute and its type are specified in the table definition.
  2. At the end of the table definition, after defining attributes we specify which attributes form the key for the table.

  For keys with one attribute, either way works.

  For keys with several attributes, #2 must be used.

- You cannot add two tuples to a table with the same key value.

- Attributes forming a key can either be defined with the constraint PRIMARY KEY or with the constraint UNIQUE.

Defining Keys: Example in Standard SQL

- First way:

```
CREATE TABLE Customers (
        custNo CHAR(10) PRIMARY KEY,
        name CHAR(100),
        born INT,
        bonus INT,
        address CHAR(100),
        email CHAR(100)
);
```

- Second way:

```
CREATE TABLE Customers
        (custNo CHAR(10),
        name CHAR(100),
        born INT,
        bonus INT,
        address CHAR(100),
        email CHAR(100),
        PRIMARY KEY (custNo)
);
```

Defining Keys: Example in SQLite

- First way:

```
CREATE TABLE Customers (
        custNo TEXT PRIMARY KEY,
        name TEXT,
        born INTEGER,
        bonus INTEGER,
        address TEXT,
        email TEXT
);
```

- Second way:

```
CREATE TABLE Customers (
        custNo TEXT,
        name TEXT,
        born INTEGER,
        bonus INTEGER,
        address TEXT,
        email TEXT,
        PRIMARY KEY (custNo)
);
```

Defining Keys with Multiple Attributes in SQLite

- If we want to define a table BelongsTo, where the order number and the product number together form a key, we have to use the 2nd way when defining the key:

```
CREATE TABLE BelongsTo(
    orderNo TEXT,
    productNo TEXT,
    count INTEGER,
    PRIMARY KEY (orderNo, productNo)
);
```

Defining Keys with Constraints in SQL

- Attributes defined with the constraint PRIMARY KEY cannot have NULL values, but attributes defined with the constraint UNIQUE can.
  - Exception in SQLite: attributes defined with the constraint PRIMARY KEY can have NULL values, if they are not disallowed due to other integrity constraints.

- If an attribute is defined with the constraint UNIQUE, many attributes of the same table can have the value NULL, even if the same values are not allowed.

Adding Tuples to the Table

- Tuples (rows) can be added to a table using the statement INSERT INTO.

For example:

INSERT INTO Products(number, prodName, description, price, manufID) **VALUES**('R–55336', 'IPad Air 2', 'tablet', 495.0, 'M554');

Here we specify the name of the table and its attributes, and then the values we want to assign for those attributes listed in the corresponding order. It is not mandatory to define a value for all attributes of the table.

If all values for all attributes are specified in the same order as the table definition, then attribute names can be left out:

INSERT INTO Products
VALUES('R–55336', 'IPad Air 2', 'tablet', 495.0, 'M554');

Adding Tuples to the Table

- All the tuples from the result of a query can be added to a table.
- Let's assume that a table for relation:

  Productgroups(name, supergroup)

  is defined for different product groups and it is empty at the moment.

  We want to add all the descriptions of the products to the names of the product groups.

```
INSERT INTO Productgroups(name)
    SELECT DISTINCT description
    FROM Products;
```

Deleting Tuples from the Table

We may delete tuples using the statement
DELETE FROM table **WHERE** condition;

This statement deletes all the tuples from the table that satisfy the condition specified in the WHERE clause.

For example: Let's delete all products with the manufacturer ID 'F542' from the table Products :

```
DELETE FROM Products
WHERE manufID = 'F542';
```

Updating Tuples in SQL

- By using the UPDATE statement we may change the values of the attributes of a tuple without needing to delete the tuple in question or add new tuples to the table.
- Syntax of the statement:

```
UPDATE table
SET assign new values
WHERE condition;
```

New values are assigned with the assignment operator (attribute = value).

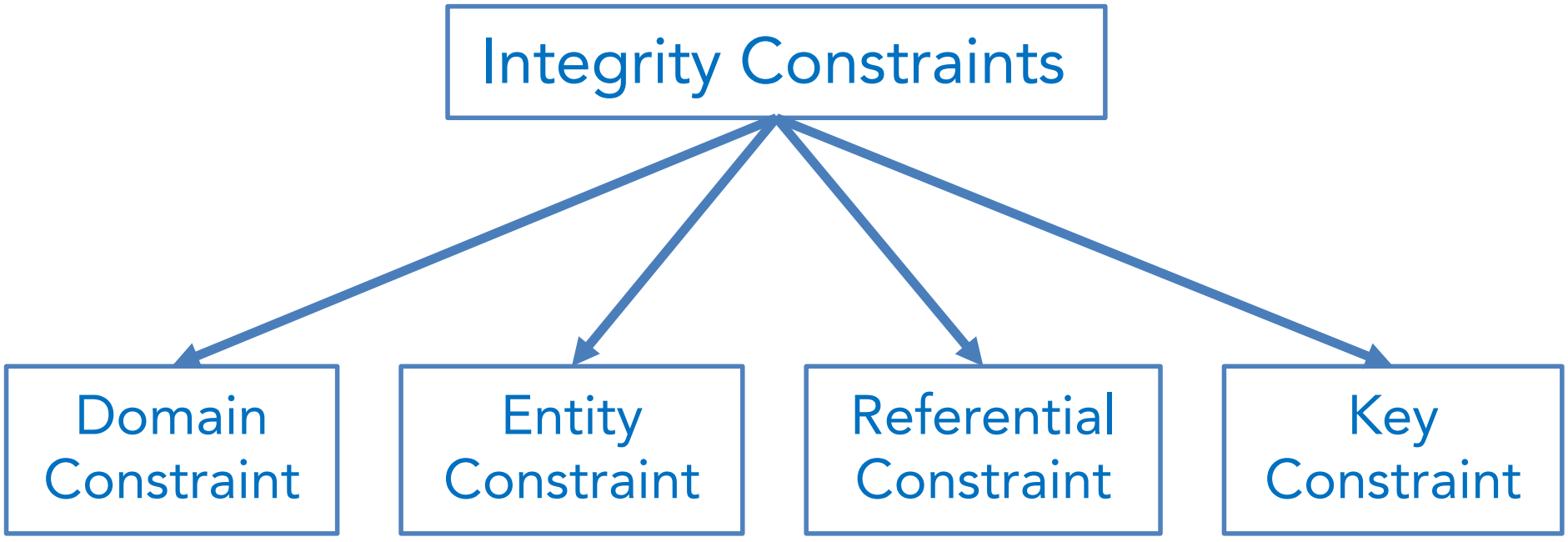
- For example: Modify description of a product to 'computer' for all the tuples currently having the description 'pc':

```
UPDATE Products
SET description = 'computer'
WHERE description = 'pc';
```

Integrity Constraints and Triggers

- SQL allows creating "active elements" that are stored in the database and executed at appropriate times to check data consistency or trigger needed actions.
- *Integrity Constraints* are logical rules that check data consistency, for instance:
  - *Key Constraints* where a set of attributes form a key for the table (i.e. two tuples cannot have the same set of values for these attributes).
  - *Referential Integrity (or foreign-key constraint)* where attributes of one table must also appear as attributes of another table.
  - SQL also allows constraints on attributes, constraints on tuples, and interrelation constraints called *Assertions*.
- Triggers are active elements that automatically runs when certain specified events happen. Triggers will be covered in the next lecture.

Integrity Constraints



Defines a valid set of values from the domain of its attribute(s) e.g. data type, default values, range, or is not Null.

Ensures that every relation has a primary key that's unique and not Null.

Defines a foreign-key constraint to maintain consistency among tuples in two relations.

Ensure two tuples cannot have the same set of values for attributes that form a key for table.

Integrity Constraints and Foreign Keys

In SQL we may declare an attribute or attributes of one table to be a *foreign key*, referencing some attribute(s) of another table. This implies the following:

- The referenced attribute(s) in the 2nd table must be declared the PRIMARY KEY or UNIQUE for the table. Otherwise, we cannot make the foreign-key declaration. E.g. Each value for the attribute manufID in the table *Products* must exist in the table *Manufacturers* as a value on the attribute ID. Here manufID is a *foreign key*.

- Values of the foreign key appearing in the first table must also appear in the referenced attributes of some tuple.

- For primary keys there's two ways to declare foreign keys:

REFERENCES <table> (<attributes>)

FOREIGN KEY (<attributes>) REFERENCES <table> (<attributes>)

Defining Foreign Keys

- We can define foreign keys in the CREATE TABLE command when listing the attributes (in SQLite):

```
CREATE TABLE Products(
        number TEXT PRIMARY KEY,
        prodName TEXT,
        description TEXT,
        price TEXT,
        manufID TEXT REFERENCES Manufacturers(ID)
);
```

In SQL standard:

```
CREATE TABLE Products(
        number CHAR(10) PRIMARY KEY,
        prodName CHAR(80),
        description VARCHAR(200),
        price REAL,
        manufID CHAR(10) REFERENCES Manufacturers(ID)
);
```

Defining Foreign Keys

- Another approach is to define the foreign keys separately with the key word FOREIGN KEY after listing the attributes:

```
CREATE TABLE Products(
        number CHAR(10) PRIMARY KEY,
        prodName CHAR(80),
        description VARCHAR(200),
        price REAL,
        manufID CHAR(10),
        FOREIGN KEY (manufID) REFERENCES Manufacturers(ID)
);
```

- If the foreign key consists of multiple attributes, we have to use the latter approach. Otherwise we can't specify that all of the values for the attributes are found together in the same tuple.

Exercise

Consider the table StarsIn (exercise round 2). Consider the difference between the definitions:

```
CREATE TABLE StarsIn(
        movieTitle CHAR(100) REFERENCES Movies(title),
        movieYear INT REFERENCES Movies(year),
        starName CHAR(30) REFERENCES MovieStar(name)
);
```

```
CREATE TABLE StarsIn(
        movieTitle CHAR(100),
        movieYear INT,
        starName CHAR(30),
        FOREIGN KEY (movieTitle, movieYear) REFERENCES Movies(title,
        year),
        FOREIGN KEY (starName) REFERENCES MovieStar(name)
);
```

Which one is correct, or are they both?

Answer

Only the latter is correct.

Why?

- The first one requires, that the name of the movie needs to exist somewhere in the table Movies and that the year of each movie needs to exist in some tuple of the table Movies, but it allows the name and the year to appear in different tuples (here it's not necessarily the same movie).

- The latter approach requires, that both the name and the year of the movie need to appear in the same tuple in the table Movies.

Maintaining Referential Integrity

- In a table *Products* with a foreign key *manufID* that refers to the attribute *ID* in table *Manufacturers*, these actions would be prevented to maintain referential integrity:
  - Inserting a new *Products* tuple with a *manufID* that is not Null and does not exist for any tuples in *Manufacturers*.
  - Updating a *Products* tuple to change *manufID* to a non Null value, but which does not exist in *Manufacturers*.
  - Removing a tuple from *Manufacturers*, but its value for *manufID* (non Null), exists as *ID* for one or more *Products*.
  - Updating a tuple in *Manufacturers* in a way that changes its *ID*, but the old value already exists as a *manufID in Products*.

Maintaining Referential Integrity

What does the DBMS do if referential integrity may be compromised due to one of the previously mentioned scenarios?

In the first 2 scenarios we restrict any modifications to integrity-breaking tuples in *Products (referring table)*. In the last two scenarios for changes to *Manufacturers (referred table)* we have 3 different options:

1. ***The Default Policy:*** *Reject Violating Modifications*: Any change violating integrity constraints in *Manufacturers rejected*.

2. ***The Cascade Policy*:** Changes to the referenced attribute(s) are mimicked at the foreign key. E.g. if we modify *Manufacturers*, we have to modify *Products* in a corresponding manner, i.e. if we delete a tuple from *Manufacturers*, we delete all those tuples from *Products*, that have *IDs* referring to *manufID*. If we update any *IDs* for *Manufacturers*, we modify the *manufIDs* of all the tuples in *Products* that refer it.

3. ***The Set-Null Policy:*** Deleting/updating value of the foreign key *manufID*, will set the value of the referring attribute *ID* to NULL.

Maintaining Referential Integrity

- We may define the policy to be used differently for deletions and updates. The policy is defined while an attribute is defined as a foreign key, for example:

```
CREATE TABLE Products(
        number CHAR(10) PRIMARY KEY,
        prodName CHAR(80),
        description VARCHAR(200),
        price REAL,
        manufID CHAR(10) REFERENCES Manufacturers(ID)
                ON DELETE SET NULL
                ON UPDATE CASCADE
);
```

The rejection policy doesn't need to be separately specified, as it's the default policy.

Constraints on the Values of Attributes

- SQL offers the possibility to set constraints on the values of attributes. The simplest constraint is to restrict the attribute from being set to the NULL value. This can be done while defining the attributes with the CREATE TABLE command *(some of the rows omitted):*

```
CREATE TABLE Products (
        ...
        price REAL NOT NULL,
        manufID CHAR(10) REFERENCES Manufacturers(ID) NOT NULL
);
```

- In this case, the DBMS doesn't allow insertions or updates, where the attributes with this constraint (in the example the attributes *price* and *manufID*) have the value NULL. We can't use the Set-Null policy with attributes that have this constraint.

Constraints on the Values of Attributes

- We can also restrict the values using the keyword CHECK.
- Two examples (*some of the rows omitted, and in the 2nd example there is a new attribute gender):*

```
CREATE TABLE Products (
        ...
        price REAL CHECK (price > 0.0 AND price < 5000.0),
        manufID CHAR(10) REFERENCES Manufacturers(ID) NOT
        NULL
);

CREATE TABLE Customers (
        ...
        gender CHAR(1) CHECK (gender IN ('F', 'M', 'O'))
);
```

Constraints on the Values of Attributes

- Alternatively we can create a domain to define a valid set of values for attributes (not supported in SQLite):

```
CREATE DOMAIN GenderDomain CHAR(1)
    CHECK (VALUE IN ('F', 'M', 'O'));
```

Now we may define tables with attributes of this domain:

```
CREATE TABLE Customers (
    ...
    gender GenderDomain
);
```

Global Constraints

- With *global constraints* we may define:
  - Constraints on different attributes of the same tuple (tuple-based *check constraint*).
  - Constraints on the tuples within a table or constraints between tables (*assertions*, *general constraints*).

- Example: let's require that in the table *Customers*, a customer can't have a name starting with 'Ms.' if the customer is not female:

```
CREATE TABLE Customers (
        custNo CHAR(10) PRIMARY KEY,
        gender CHAR(1),
        ....
        CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

Global Constraints

- When defining the table *R*, the tuple-based *check constraint* may contain any conditions that can appear in the WHERE part of an ordinary query, including subqueries.
- If the subquery of the constraint includes another table *S*, check the constraint only when modifying *R*, not when modifying *S*.
- Example *(not supported by SQLite):* let's define a constraint on the table *Products*, that no manufacturer can have more than 10,000 euros worth of products in the store:

```
CREATE ASSERTION SumPrice CHECK (10000.0 >= ALL
      (SELECT SUM(price) FROM Products GROUP BY manufID)
);
```

- As this constraint is only for tuples of a single table, it can be defined as a tuple-based check constraint when creating the table.
- Currently most DBMS (including SQLite) don't support queries in the CHECK clause, although the SQL standard allows them.

Modifying the Constraints

- We can modify a constraint only if it's given a name when created.
- We can name a constraint by adding the keyword CONSTRAINT when defining it, and writing a name after that. Examples:

```
CREATE TABLE Products (
        ...
        price REAL CONSTRAINT CorrectPrice
                        CHECK (price > 0.0 AND price < 5000.0)
);

CREATE TABLE Customers (
        custNo CHAR(10) CONSTRAINT NumberKey PRIMARY KEY,
        gender CHAR(1),
        ....
        CONSTRAINT RightTitle
                CHECK (gender = 'F' OR name NOT LIKE 'Ms.%')
);
```

Modifying the Constraints

- We may modify the constraints using the keywords ALTER TABLE and simply DROP or ADD *(this is not supported in SQLite).*
- Examples:

```
ALTER TABLE Customers DROP CONSTRAINT
NumberKey;

ALTER TABLE Customers ADD CONSTRAINT
    NumberKey PRIMARY KEY (custNo);

ALTER TABLE Customers DROP CONSTRAINT RightTitle;
```

- Constraints created with the CREATE ASSERTION command can be deleted with the command DROP ASSERTION, for example:

```
DROP ASSERTION SumPrice;
```

Views

- While we can create tables that are persistently stored in a database, we can also define **views** created as "virtual tables" that are temporarily created by query expressions.
- A view can be defined with the command:

  CREATE VIEW name **AS** definition;

  where name is the name of the view we are defining, and definition is some SQL query.
- Example: let's create a view, that contains only the products with the manufacturer "Samsung" from *Products*:

```
CREATE VIEW SamsungProducts AS
    SELECT number, prodName, description, price, manufID
    FROM Products, Manufacturers
    WHERE manufName = 'Samsung' AND manufID = ID;
```

Views: Example

- Example: let's create a view with the number and name of a product combined with the name of the manufacturer:

```
CREATE VIEW ProdManuf AS
    SELECT number, prodName, manufName
    FROM Products, Manufacturers
    WHERE manufID = ID;
```

- We can target queries on views in a same manner as with ordinary tables. We can use both ordinary tables and views in the same query.
- Example: search for order numbers of orders that contain products from Samsung *(using a view created from Products):*

```
SELECT DISTINCT orderNo
FROM SamsungProducts, BelongsTo
WHERE number = productNo;
```

Using Views

- With a view we can simplify the queries we want to write, as part of the query is in a sense hidden behind the definition of the view.

- Using a view doesn't make the query more efficient, as the view is created again (the query defining the view will be executed) whenever the view is used in a query.

- In addition, we can use views to manage the permissions of the users of the database. We can define, that a user can use a restricted set of views, but not access the original tables.

Renaming Attributes

- While defining a view, we may rename the attributes inside parenthesis after the name of the view.
- Example:

```
CREATE VIEW ProdManuf(productNumber,
productName, ManufacturerName)
SELECT number, prodName, manufName
FROM Products, Manufacturers
WHERE manufID = ID;
```

- This view is identical to the view in the earlier example, but here the attributes have different names.

Note: This does not seem to work in SQLite when tested, even though it should, based on the documentation.

Modifying Views

- We can remove a view with the command DROP VIEW, for example:

  DROP VIEW ProdManuf;

  This command only deletes the definition of the view, not the tables used to create the view.

- It's possible to write SQL queries, that dynamically update the view (insert, delete or update tuples of the view), though the SQL rules for creating *Updatable Views* are complex. However, once created they can be queried and modified like regular views.
- Views can be *materialized*, which means storing the view and updating it whenever it's necessary. In this course, we won't cover views as thoroughly, but in the next lecture we will cover indexes which are a form of materialized views.