INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2021/22

THE RELATIONAL DATA MODEL

Relational Data Model
In the relational data model, data is exclusively organized in **relations**, i.e., **sets of tuples** of data. Data in each **attribute** (tuple component) is **atomic** and of a declared **type**.

Relational Database Management System
A relational database management system (short: **RDBMS**) implements the relational data model. RDBMSs provide persistent storage for relations (as well as auxiliary data structures).

- We will use **PostgreSQL** as a typical member of the family of contemporary RDBMSs. The following code examples use the system's REPL psql (version 13).

USER DATA IN RELATIONS

- **User data** is organized in relations (here: relation colors of the LEGO sets mini-world):

```
lego=# table colors;
+-------+------------------------+-------------+--------+-----------+---------+
| color |          name          |   finish    |  rgb   | from_year | to_year |
+-------+------------------------+-------------+--------+-----------+---------+
|     0 | (Not Applicable)       | N/A         | □      |      1954 |    2013 |
|    41 | Aqua                   | Solid       | b5d3d6 |      1998 |    2006 |
|    11 | Black                  | Solid       | 212121 |      1957 |    2013 |
|     7 | Blue                   | Solid       | 0057a6 |      1950 |    2013 |
|    97 | Blue-Violet            | Solid       | 506cef |      2004 |    2005 |
|    36 | Bright Green           | Solid       | 10cb31 |      1950 |    2013 |
|   105 | Bright Light Blue      | Solid       | 9fc3e9 |      2004 |    2013 |
|   110 | Bright Light Orange    | Solid       | f7ba30 |      2000 |    2013 |
|   103 | Bright Light Yellow    | Solid       | f3e055 |      2004 |    2013 |
|   104 | Bright Pink            | Solid       | ffbbff |      2003 |    2013 |
|     8 | Brown                  | Solid       | 532115 |      1974 |    2006 |
    ⋮
```

META DATA IN RELATIONS

- The *everything is a relation* principle is further applied to **RDBMS-internal data** (or: **database catalog**).
  - Example: list relations that model the LEGO mini-world (psql command \dt):

```
lego=# \dt
              List of relations
+--------+--------------+-------+-------+
| Schema |     Name     | Type  | Owner |
+--------+--------------+-------+-------+
| lego   | available_in | table | grust |
| lego   | bricks       | table | grust |
| lego   | categories   | table | grust |
| lego   | colors       | table | grust |
     ⋮
| lego   | contains     | table | grust |
| lego   | minifigs     | table | grust |
| lego   | pieces       | table | grust |
| lego   | sets         | table | grust |
+--------+--------------+-------+-------+
```

META DATA IN RELATIONS

- **More meta data (data about data):** Information about **attributes** and their data **types** for user relation colors. Use psql command \d *t* (*describe relation t*):

```
lego=# \d colors
                          Table "lego.colors"
+-----------+-----------------------+-----------+----------+---------+
|  Column   |         Type          | Collation | Nullable | Default |
+-----------+-----------------------+-----------+----------+---------+
| color     | integer               |           | not null |         |
| name      | character varying(30) |           |          |         |
| finish    | character varying(15) |           |          |         |
| rgb       | rgb                   |           |          |         | ⚠
| from_year | integer               |           |          |         |
| to_year   | integer               |           |          |         |
+-----------+-----------------------+-----------+----------+---------+
```

- Focus on attributes Column, Type (will address Nullable, Default later).
- Note: Type rgb appears to be rather specific for the LEGO sets mini-world.

TYPES AND DOMAINS

- **Meta data:** Data **types** available in PostgreSQL (psql command \dTS):

```
lego=# \dTS
                                List of data types
+------------+---------------+----------------------------------------------+
|   Schema   |     Name      |                 Description                  |
+------------+---------------+----------------------------------------------+
| lego       | id            |                                              | ⚑
| lego       | rgb           |                                              | ⚑
| lego       | type          |                                              | ⚑
| pg_catalog | boolean       | boolean, 'true'/'false'                      |
| pg_catalog | integer       | -2 billion to 2 billion integer, 4-byte storage |
| pg_catalog | bytea         | variable-length string, binary values escaped |
| pg_catalog | json          |                                              |
| pg_catalog | point         | geometric point '(x, y)'                     |
| pg_catalog | money         | monetary amounts, $d,ddd.cc                  |
       ⋮
```

- Types marked ⚑ added by the user, all others built into PostgreSQL 13.

TYPES AND DOMAINS

Types

Let $\mathbb{T}$ denote **the set of all data types** (built-in and user-defined). Any value stored in a relation cell must be of a type $t \in \mathbb{T}$. When PostgreSQL starts, $\mathbb{T}$ is initialized as

$$\mathbb{T} = \{ \text{boolean}, \text{integer}, \text{text}, \text{bytea}, \dots \}.$$

(See rows with pg_catalog in column Schema in output of command \dTS.)

Values

Any value stored in a relation cell is an element of the **set of all values** $\mathbb{V}$. In the relational data model, all values $v \in \mathbb{V}$ are atomic:

$$\mathbb{V} = \{ \text{true}, \text{false}, 0, -1, 1, -2, 2, \dots \}.$$

TYPES AND DOMAINS

Domains

For any $t \in \mathbb{T}$, its **domain** $dom(t)$ is the set of all values of type t. $dom(\cdot)$ thus is a function with signature $\mathbb{T} \to 2^{\mathbb{V}}$. For example:

$$dom(\text{integer}) = \{\ 0, -1, 1, -2, 2, \ldots\ \}$$
$$dom(\text{boolean}) = \{\ \text{true}, \text{false}\ \}$$

- Any value $v \in \mathbb{V}$ has one of the admissible types in $\mathbb{T}$: v **has type** $t \Leftrightarrow v \in dom(t)$.
- PostgreSQL is extensible: users can add user-defined types and values to $\mathbb{T}$ and $\mathbb{V}$ (and thus also alter $dom(\cdot)$), respectively.

TYPES AND DOMAINS

- The domain of the generic built-in types like integer or text (variable-length strings) is often **too large** to precisely model mini-worlds.
  - Example: Modeling a person's age by type integer admits non-sensical values like −1 or 500.

CREATE DOMAIN

The SQL command CREATE DOMAIN creates a new type *t'* based on an existing type *t* with $dom(t') \subseteq dom(t)$. Constraints may be provided that define the admissible values *v* of *t'*:

```
CREATE DOMAIN t' [ AS ] t
  [ CHECK (expression) ]
```

In Boolean expression *expression*, use name VALUE to refer to *v*.

INTERLUDE: POSTGRESQL DOCUMENTATION

- For most SQL (and psql) commands we will only discuss those aspects that are relevant in the context of this course. The gory details are to be found in PostgreSQL's own (excellent!) documentation:

1. For a brief overview inside the psql REPL:

```
=> \h ‹SQL command›
=> \?
```

2. Full documentation (Web): `https://www.postgresql.org/docs/13/index.html` (use Search Documentation)

- Documentation (and slide) conventions:
  - CREATE DOMAIN: literal syntax
  - *t*: variable parts of the syntax (on slides sometimes: ‹*t*›)
  - [ ]: optional parts of command syntax, { | }: alternative parts, …: repeatable parts

TYPES AND DOMAINS

Examples: CREATE DOMAIN

```
-- Create new type called `rgb': strings of exactly six hex digits (RGB color rrggbb)
-- (operator ~ denotes regular expression matching)
CREATE DOMAIN rgb AS text
  CHECK (VALUE ~ '^(\d|[a-f]){6}$');

-- Create new type called `type': the single character 'B' or 'M'
-- (operator IN checks for the presence of an element in a list of values)
CREATE DOMAIN type AS character(1)
  CHECK (VALUE IN ('B', 'M'));              -- (B)rick or (M)inifigure

-- Create new type called `id': alias for the built-in type of strings of max length
20
CREATE DOMAIN id AS
  character varying(20);
```

- Note: *dom*(rgb) ⊆ *dom*(text), *dom*(type) ⊆ *dom*(character(1)), and *dom*(id) = *dom*(character varying(20)).

TYPES AND DOMAINS

- `CREATE DOMAIN` ...
  1. ... establishes a **single place** where mini-world specific types are defined,
  2. ... can **introduce mnemonic names** for non-descriptive, generic type names.
- `CREATE DOMAIN` t' … inserts t' into set $\mathbb{T}$ of all types (see types in schema `lego` in `\dTS` output above).
- `CREATE DOMAIN` t' … also defines $dom(t')$. Current state of $dom(\cdot)$ after the domains on the last slide have been created (`psql` command `\dD`):

```
lego=# \dD
                                 List of domains
+--------+------+--------------------+----------------------------------------+
| Schema | Name |        Type        |                 Check                  |
+--------+------+--------------------+----------------------------------------+
| lego   | id   |character varying(20)|                                       |
| lego   | rgb  |text                |CHECK (VALUE ~ '^(\d|[a-fA-F]){6}$'::text)|
| lego   | type |character(1)        |CHECK (VALUE = ANY (ARRAY['B', 'M']))   |
+--------+------+--------------------+----------------------------------------+
```

INTERLUDE: EVALUATION OF SQL EXPRESSIONS

- The SQL command

```
SELECT expression [ AS output_name ] [, …]
```

  evaluates *expression* and returns a single-row relation whose cells contain the results. If specified, attributes are named *output_name* (otherwise the name of the expression's type or "?column?" are used as attribute names).

- Example (Note: in psql, SQL commands are terminated by a semicolon ;):

```
=> SELECT 1904 > 1893 AS result;
+--------+
| result |
+--------+
| t      |
+--------+
```

TYPES AND DOMAINS

- SQL derives the types of expressions automatically, but an explicit **type cast** operation may be used to enforce that an expression has a given type *t* (if possible at all):

Type Cast

A **type cast** converts the type *t* of an expression to type *t'* if this is allowed by the SQL type conversion rules. These are equivalent:

```
CAST (expression AS t')
expression :: t'
```

- Casting a string literal to some type *t'* always succeeds if the string contents are acceptable syntax for a value of type *t'* (OK: `'42' :: integer`, `'true' :: boolean`, `'f' :: boolean` — fails: `'4+2' :: integer`, `'x' :: boolean`)

TYPES AND DOMAINS

- The RDBMS **enforces the domain constraints** during expression evaluation:

```
lego=# SELECT 'ff00ff' :: rgb, 'B' :: type;
+--------+-------+
|  rgb   | type |
+--------+-------+
| ff00ff | B     |
+--------+-------+
lego=# SELECT 'foobar' :: rgb;
ERROR:  value for domain rgb violates check constraint "rgb_check"
```

- If *t'* derives from *t* via CREATE DOMAIN, values of these types are **compatible:**

```
lego=# SELECT 'M' :: type = 'X', '#' || ('ff00ff' :: rgb);
+----------+----------+
| ?column? | ?column? |
+----------+----------+
| f        | #ff00ff  |
+----------+----------+
```

TYPES AND DOMAINS

- This compatibility between types is often convenient but also carries the risk of confusion (in complex applications): "*comparing apples with oranges*" remains possible.

CREATE TYPE

SQL command CREATE TYPE creates a new type t' that is **distinct from any other type.** We have $dom(t') \cap dom(t) = \emptyset$ for all other types t:

```
CREATE TYPE t' AS ENUM
  ( [ 'label' [, …] ] )
```

The domain of **enumerated** type t' is $dom(t') = \{\text{'}label_1\text{'}, \text{'}label_2\text{'}, \ldots\}$.

- Note: The CREATE TYPE command extends $\mathbb{T}$ and $dom(\cdot)$ as well as adds new elements to $\mathbb{V}$.

TYPES AND DOMAINS

Examples: CREATE TYPE

```
-- The five days of the working week (starts Monday, ends Friday)
CREATE TYPE workday AS ENUM ('Mon', 'Tue', 'Wed', 'Thu', 'Fri');

-- Is this LEGO piece a (B)rick or a (M)inifigure? (cf. type `type' above)
CREATE TYPE brick_minifig AS ENUM ('B', 'M');
```

- The RDBMS knows about the new (explicitly enumerated) domain and rejects values outside that domain:

```
=> SELECT 'Sat' :: workday;
ERROR:  invalid input value for enum workday: "Sat"
LINE 1: SELECT 'Sat' :: workday;
```

TYPES AND DOMAINS

- The enumeration of the type’s domain implicitly **defines an order** on its values:

```
=> SELECT 'Wed' :: workday < 'Thu' :: workday, 'Wed' < 'Thu';
+----------+----------+
| ?column? | ?column? |
+----------+----------+
| t        | f        |
+----------+----------+
```

- Since the **domain of the new type is disjoint from any other**, comparison with values of other types are not admitted:

```
=> SELECT 'Mon' :: workday = 'Mon' :: text;
ERROR:  operator does not exist: workday = text
LINE 1: SELECT 'Mon' :: workday = 'Mon' :: text;
                                ^
```

SCHEMATA AND RELATIONS

- In the relational data model, each **attribute** of a table has a **declared type:**

```
lego=# \d colors
               Table "lego.colors"
+-----------+-----------------------+-----------+----------+---------+
|  Column   |         Type          | Collation | Nullable | Default |
+-----------+-----------------------+-----------+----------+---------+
| color     | integer               |           | not null |         |
| name      | character varying(30) |           |          |         |
| finish    | character varying(15) |           |          |         |
| rgb       | rgb                   |           |          |         |
| from_year | integer               |           |          |         |
     ⋮
```

- If an attribute has declared type t, the RDBMS will exclusively store values v in that attribute such that

1. $v \in dom(t)$ or
2. v can be successfully cast to type t.

SCHEMATA AND RELATIONS

Attributes (Columns)
Let $\mathbb{A}$ denote the set of **attribute names** of *all* relations.
Attribute Types
Any attribute $a \in \mathbb{A}$ has a declared **(attribute) type** $type(a) = t \in \mathbb{T}$ (i.e., $type(\cdot)$ is a function with signature $\mathbb{A} \to \mathbb{T}$).

- Once the type of a is declared, its set of admissible values is known:

Attribute Values
The set of **(admissible) attribute values** for attribute a is

$$val(a) := dom(type(a)).$$

$val(\cdot)$ thus is a function with signature $\mathbb{A} \to 2^{\mathbb{V}}$.

SCHEMATA AND RELATIONS

- Attributes are introduced and their types declared whenever we **create a new relation.**

CREATE TABLE

The SQL command CREATE TABLE creates a new relation t with specified names and types (the column names of t must be unique):

```
CREATE TABLE t (
   [ column_name type [, …] ]
)
```

- The CREATE TABLE command introduces new typed attributes and thus affects set $\mathbb{A}$ of all attributes and type assignment $type(\cdot)$.

SCHEMATA AND RELATIONS

Example: CREATE TABLE

```
CREATE TABLE minifigs (
  piece  id,               -- unique piece identifier assigned by LEGO
  type   type,             -- = 'M' (this is a minifig)
  name   text,             -- human-readable minifig name
  cat    integer,          -- category (LEGO theme) this minifig is part of
  weight real,             -- in g
  img    text              -- URL pointing to piece's image at BrickLink.com
);
```

- After this CREATE TABLE command, for example:
  - $\mathbb{A}$ = { piece, type, name, cat, weight, img, … }
  - *type*(piece) = id, *type*(img) = text
  - *val*(type) = *dom*(*type*(type))= { 'B', 'M' }

SCHEMATA AND RELATIONS

Relation Schema

A **relation schema** associates a relation name R with its set of declared attributes (a subset of $\mathbb{A}$):

$$(R, \{a_1, \ldots, a_n\})$$

Common notation: $R(a_1, \ldots, a_n)$. R is called n-ary relation. Also: $sch(R) = \{a_1, \ldots, a_n\}$ and $deg(R) = n$ (degree).

Relational Database Schema

A non-empty finite set of relation schemata makes a **relational database schema**

$$\{(R_1, \alpha_1), (R_2, \alpha_2), \ldots\}$$

where $\alpha_i \subseteq \mathbb{A}$. In a relational database schema, the relation names R_i are unique.

INTERLUDE: MULTIPLE SCHEMATA

- In a database schema, relation names (as well as the names of domains, types, and further database objects) are assumed to be unique.
- RDBMSs support **multiple schemata to partition the available namespace** to reduce naming conflicts/collisions.

CREATE SCHEMA

A new namespace partition is introduced via

```
CREATE SCHEMA schema_name
```

A database object (table, type, domain, …) named *t* in that new partition can be accessed by its **qualified name** *schema_name.t* .

INTERLUDE: MULTIPLE SCHEMATA

- In PostgreSQL, when an unqualified name *t* is used, the **current schema** is used to construct a fully qualified name:

```
=# SELECT current_schema();
```

current_schema
public

- The current (default) schema can be set (initially set to 'public'):

```
=# set schema 'lego';
=# SELECT current_schema();
```

current_schema
lego

SCHEMATA AND RELATIONS

- Now changing focus from relation schema (heading) to relation contents (body).

Tuple

Given a relation $R(a_1, \ldots, a_n)$, a **tuple** t **of** R maps attributes to values, i.e., t is a function with signature $\{a_1, \ldots, a_n\} \rightarrow \mathbb{V}$ with

$$\forall\, a \in \{a_1, \ldots, a_n\}: t(a) \in val(a)$$

Common notation for $t(a)$ is $t.a$ ("dot notation", "t dot a", "the a-value of t").

- Recall: $val(a) := dom(type(a))$.

SCHEMATA AND RELATIONS

- Note that this understanding of tuples t coincides with our PyQL representation of rows as Python dictionaries:

$$t(a_1) = v_1,\ \ldots,\ t(a_n) = v_n \equiv \{\ a_1 : v_1,\ \ldots,\ a_n : v_n\ \}$$

- In these dictionaries, the order of key/value pairs is immaterial. Attribute access is entirely name-based and position-independent.
  In particular: two tuples t, t' over $\{a_1, \ldots, a_n\}$ are equal if

$$\forall\, a \in \{a_1, \ldots, a_n\}: t(a) = t'(a)$$

  - Python:

```
>>> {'a': 42, 'b': 'LEGO', 'c': False} == {'c': False, 'a': 42, 'b': 'LEGO'}
True
```

RELATION STATE

- Once declared, a **relation schema very seldomly changes** in typical applications of database systems.
- However, the **set of tuples stored in a relation is expected to frequently change.**

Relation Instance (State)
The current finite *set* of tuples t_i of relation $R(a_1, \ldots, a_n)$ is called the relation's **instance** (or: **state**)

$$inst(R) = \{t_1, t_2, \ldots, t_m\}$$

Database (Instance) State
The **database instance** comprises the instances of all its relations.

- NB: Since $inst(R)$ is a *set* of tuples, all t_i are mutually different. This matches the 1970 formulation of Edgar F. Codd's relational model[1] .

CODD'S RELATIONAL DATA MODEL

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.

Existing noninferential, formatted data systems provide users with tree-structured files or slightly more general network models of the data. In Section 1, inadequacies of these models are discussed. A model based on *n*-ary relations, a normal form for data base relations, and the concept of a universal data sublanguage are introduced. In Section 2, certain operations on relations (other than logical inference) are discussed and applied to the problems of redundancy and consistency in the user's model.

KEY WORDS AND PHRASES: data bank, data base, data structure, data organization, hierarchies of data, networks of data, relations, derivability, redundancy, consistency, composition, join, retrieval language, predicate calculus, security, data integrity
CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22, 4.29

Communications of the ACM, 13(6), June 1970

SQL VS. THE RELATIONAL DATA MODEL ⚠️

- Codd's relational model is the foundation on which SQL RDBMSs have been built since the 1970s. **SQL database systems deviate from their relational roots** in important aspects, however.
  - The clean formal foundation has led to the most efficient and versatile database systems technology that is available to date.
  - The deviation has led to confusion and "religious wars" about what constitutes a *true relational database system*. (If you are interested, search for books and articles by Chris J. Date and Hugh Darwen.)
- Database languages and systems closely tracking Codd's original relational model are *Tutorial D* and *Rel*, respectively (see `http://www.thethirdmanifesto.com`).
- Here, we will primarily stick with **SQL** (but also discuss aspects of the original relational model). Measures have been taken to keep confusion in check.

SQL VS. THE RELATIONAL DATA MODEL

Rows vs. Tuples

- The SQL CREATE TABLE command prescribes an *order* of the attributes of a relation. This deviates from the tuple model (name-to-value mapping).

Row

Given a SQL table $R(a_1, \ldots, a_n)$, a **row** t **of** R is an ordered sequence (a_i is called the i-th **column**)

$$t = (v_1, \ldots, v_n) \in val(a_1) \times \cdots \times val(a_n)$$

t is a function $\{1,\ldots,n\} \to \mathbb{V}$ with $\forall\, i \in \{1,\ldots,n\}\colon t(i) \in val(a_i)$.

- Nevertheless, SQL refers to attributes by name (positional attribute access is supported by some SQL constructs, e.g., GROUP BY or ORDER BY → later).

SQL VS. THE RELATIONAL DATA MODEL

Table State vs. Relation State

Table Instance (State)

The contents of a SQL table R is a finite **unordered list** (or: multiset) of rows. In particular, a table may contain **duplicate rows** (if constraints do not say otherwise).

- ⚠️ There is *no* first or last row in a table. (Since a defined order of rows can be helpful for humans and/or applications, SQL supports row ordering when the final result of a query is returned.)
- Row equality: $(v_1, \dots, v_n) = (v_1', \dots, v_n') \Leftrightarrow v_1 = v_1' \land \cdots \land v_n = v_n'$.

SQL VS. THE RELATIONAL DATA MODEL

Table State vs. Relation State

- The current state of a table (i.e., its multiset of tuples) may be queried at any time:

TABLE

Query the current state of table *t*.

```
TABLE t
```

- Right after table creation: $inst(t) = \emptyset$ and the output of `TABLE t` will be empty.
- ⚠️ Since the table state is a *multiset* of tuples, `TABLE t` will list the tuples in *some* order (tuple “insertion order” is immaterial, in particular).

SQL VS. THE RELATIONAL MODEL

Concept/Terminology Summary

CSV	Relational Model	SQL
–	Domain	Domain
–	Type	Type
–	Schema	Schema
File	Relation	Table
Line	Tuple	Row
Field	Attribute	Column

- You will find that textbooks, papers, practitioners, academics, these slides, and even PostgreSQL use a mixture of terminology. Deal with it.

MODIFYING TABLE STATE

- Up to here, we have focused on the **Data Definition Language (DDL)** built into SQL (define domains, types, create table schemata).
- SQL's **Data Manipulation Language (DML)** modifies and queries table states.

INSERT

The SQL DML command INSERT adds rows to the state of table *t*. Order and types of the inserted values must match the specified column list. Columns not present in the list (but in *t*) are filled with the special SQL **NULL** value.

```
INSERT INTO t (column_name [, …])
  VALUES (expression [,…]) [, …]
```

- Note: multiple rows can be added with one INSERT command.

MODIFYING TABLE STATE

DELETE

The SQL DML command DELETE removes those rows from the state of table *t* that satisfy the given *condition*, i.e., a Boolean expression (or: predicate) that can refer to the individual rows of *t*:

```
DELETE FROM t [ AS ] alias
  [ WHERE condition ]
```

- Inside *condition*, the current row of *t* may be referred to by the (user-defined) name *alias*. Column *c* of that row thus is referred to as *alias.c* (dot notation).
- A missing WHERE clause is interpreted as *condition* ≡ TRUE. (Also see PostgreSQL's TRUNCATE command.)

MODIFYING TABLE STATE

UPDATE

The SQL DML command UPDATE modifies column values of existing rows in table *t*. Predicate *condition* identifies those rows that will be affected. Columns not referred to in the SET clause remain unchanged.

```
UPDATE t [ AS ] alias
  SET column_name = expression [, …]
  [ WHERE condition ]
```

- The new column values typically depend on the old (existing) values. In *expression* (and *condition*), *alias* may be used to refer the old row of *t*.

MODIFYING TABLE STATE

- More ways to add to table states:

1. Use a SQL query to **compute the rows to be added** to table *t*:

```
INSERT INTO t (column_name [, …])
  { VALUES (expression [, …]) [, …] | query }
```

2. Use `COPY … FROM …` to **load contents from CSV files** (see above).

3. Use PostgreSQL's **foreign data wrapper** such that the state of table *t* *directly* reflects the contents of a regular CSV file.

   - ⚠️ Advanced (although simple). Relies on PostgreSQL extension `file_fdw`.
   - Currently read-only. SQL DML operations on *t* do not alter the CSV file.

LITERAL TABLES

- A **literal table** (or: table constant) may be constructed by explicitly listing the rows in the table's state:

> **VALUES**
>
> The SQL VALUES expression constructs a **literal table** containing the rows specified. All rows must contain the same number of columns.
>
> ```
> VALUES (expression [, …]) [, …]
> ```
>
> The resulting table is unnamed, its columns are automatically named "column1", "column2", …. Column types are inferred from the column contents.

- A table constructed by VALUES is essentially anonymous (just like the integer literal 42 has no name either). Explicit table and column (re)naming can be performed in the context of SQL SELECT queries (→ soon).

1. Codd, E.F. “A Relational Model of Data for Large Shared Data Banks”. Communications of the ACM 13 (6): 377–387. ↩