INTRODUCTION TO

RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2021/22

DATABASE DESIGN

- Given a particular mini-world, almost always will there be plenty of options on how to choose
  - column data types,
  - table schemata, and
  - relationships between tables (e.g., foreign keys).
- The upcoming material discusses **table and database design** options, and introduces
  - **relational normal forms** that measure the redundancy of a given table design, and
  - the **Entity Relationship (ER)** model that translates a graphical sketch of a mini-world into table designs.
- Along the way, we will pick up plenty of further SQL constructs, some basic, some advanced.

ATOMIC VALUES IN TABLE CELLS

- The relational data model is **flat:** table cell values are **atomic.** Be more precise now.

> **Atomic Values, First Normal Form**
> We regard a value *v* as being **atomic** if *v* does *not* possess a tabular structure.
>
> A table whose cell values are atomic is in **First Normal Form (1NF).**

- Under this definition …

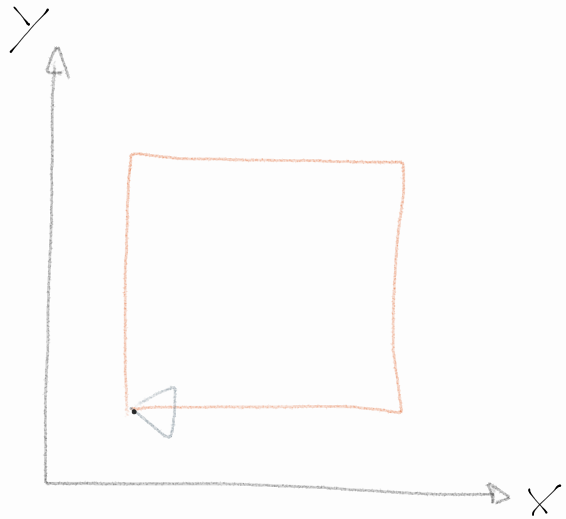
1. … is a string (e.g., of type `text`) value in a table cell atomic?
2. … is a value of type `date` (with day, month, year components) atomic?
3. … is a value of a row type atomic?
4. … is an array of type *t*[] (with type *t* being atomic) atomic?
5. … is a table nested inside a table cell atomic?

(STRUCTURED) TEXT IN TABLE CELLS

- Use column turtle of type text to hold **a list** of Logo-style drawing commands.
  Text encoding of drawing commands: 'p,x,y; …': put pen up/down (p ∈ {u,d}), then move pen by x units right and y units up across paper.

shapes

id	shape	turtle
1	square	'd,0,10; d,10,0; d,0,-10; d,-10,0'
2	triangle	'd,5,10; d,5,-10; d,-10,0'
3	cross	'd,0,10; u,-5,-5; d,10,0'
⋮	⋮	⋮



(STRUCTURED) TEXT IN TABLE CELLS

- If *r* is a row of table `shapes`, SQL DML commands can use *r*.`turtle` to **access the entire string of drawing commands** in SQL expressions. From the viewpoint of SQL, column `turtle` is atomic.
- PostgreSQL's library of string functions and operators can access selected individual parts of the string: `https://www.postgresql.org/docs/current/functions-string.html`.
- To access the list of individual drawing command either requires

1. **PostgreSQL-specific support** for regular expression matching (e.g., `regexp_split_to_table()`: return a table of substrings, i.e., generate a tabular structure that is accessible for SQL), or
2. an **iterative or recursive SQL query** that chops off the leading '*p*,*x*,*y*;' triple until the drawing command string is empty.

- Both options are awkward and inefficient.
- ⚠️ **Encoding structured content in `text` cells** is (all too) common but **definitely bad table design practice.** Interesting and relevant mini-world structure is hidden from SQL.

ARRAYS IN TABLE CELLS

- For any type *t* (including the user-defined types, e.g., composite types), PostgreSQL also supports *t*[], its associated **array type**. All elements of a *t*[] array are of type *t*:

```
ARRAY[v₁ :: t, v₂ :: t, …]          -- array of t elements, printed as {v₁,v₂,...}
ARRAY[] :: t[]                      -- empty array of t elements, printed as {}
```

- **Accessing array xs:**

```
xs[i]                               -- indexed access, i ⩾ 1 (NULL if outside bounds)
xs[i:j]                             -- array slice
```

- **Array operations:**

```
=, <>, <, >                            -- array to array comparison
expression {=|<|>|…} {ANY|ALL}(xs)     -- element to array comparison
@>, <@, &&                             -- contains, is contained by, overlaps
||                                     -- concatenation
```

ARRAYS IN TABLE CELLS

- Encode the list of turtle drawing commands in terms of

1. user-defined row type `(down boolean, x integer, y integer)` named `cmd`, and
2. column `turtle` of array type `cmd[]`:

shapes

id	shape	turtle
1	square	{(t,0,10), (t,10,0), (t,0,-10), (t,-10,0)}
2	triangle	{(t,5,10), (t,5,-10), (t,-10,0)}
3	cross	{(t,0,10), (f,-5,-5), (t,10,0)}

- Access the individual elements of an array via PostgreSQL's table-generating function `unnest()`. Function call `unnest(ARRAY[`v_1, v_2, v_3, …`])` yields

v_1
v_2
v_3
⋮

TABLES IN TABLE CELLS

- **Recursively** apply the idea of structuring information in tabular form: use a **nested table** to represent the turtle drawing command lists. We end up with a table in **Non-First Normal Form (NFNF, NF^2).**

shapes

id	shape	turtle	
		pos	command
1	square	1	(t,0,10)
		2	(t,10,0)
		3	(t,0,-10)
		4	(t,-10,0)
		pos	command
2	triangle	1	(t,5,10)
		2	(t,5,-10)
		3	(t,-10,0)
		pos	command
3	cross	1	(t,0,10)
		2	(f,-5,-5)
		3	(t,10,0)

TABLES IN TABLE CELLS (NF²)

- Notes:

1. Column pos encodes command order (list semantics) in the nested tables.
2. Outer table shape has 3 rows. Type of turtle: table(pos int, command cmd).
3. NF² admits recursion to arbitrary depth. "NF² SQL" queries reflect this recursion:

```
-- Find shapes drawn with multiple strokes
SELECT s.id, s.shape
FROM   shapes AS s
WHERE  EXISTS (SELECT 1
               FROM   s.turtle AS c              -- s.turtle has type table(…)
               WHERE  NOT (c.command).down);
```

4. ⚠️ **No off-the-shelf RDBMS supports the NF² model** (mostly a 1980s research idea). Still a powerful/modular way to think about data modelling.

- Possible: Systematic (algorithmic) conversion of any NF² table into (a bundle of) equivalent 1NF tables.

FROM NF² TO 1NF

```
nf2to1nf(R)  (input: table R, output: a table bundle of size ≥ 1):

  for each a ∈ sch(R) do
    if type(a) = table(b₁ t₁, …, bₖ tₖ, …, bₘ tₘ) then                 } see ⊡
      Create a new table Rₐ(a surrogate, b₁ t₁, …, bₖ tₖ, …, bₘ tₘ)    } below
      for each row r ∈ inst(R) do
        Create a new value τ of type surrogate
        if table r.a is not empty then
          for each row (v₁, …, vₘ) ∈ r.a do
            Insert row (τ, v₁, …, vₘ) into Rₐ
        Set r.a to τ                } if r.a is empty, τ will not have a match in Rₐ
      Set type(a) to surrogate
      nf2to1nf(Rₐ)
```

Notes:

- ⊡: If $\{b_1, \dots, b_k\}$ is the key of the table nested in column a, the key of new table R_a will be $\{a, b_1, \dots, b_k\}$.

FROM NF² TO 1NF

- Result of **nf2to1nf**(shapes), shapes.turtle refers to turtles.turtle (⚠ not a FK):

shapes (R)

id	shape	turtle
1	square	τ_1
2	triangle	τ_2
3	cross	τ_3

turtles (R_{turtle})

turtle	pos	command
τ_1	1	(t,0,10)
τ_1	2	(t,10,0)
τ_1	3	(t,0,-10)
τ_1	4	(t,-10,0)
τ_2	1	(t,5,10)
τ_2	2	(t,5,-10)
τ_2	3	(t,-10,0)
τ_3	1	(t,0,10)
τ_3	2	(f,-5,-5)
τ_3	3	(t,10,0)

FROM NF² TO 1NF

The surrogate-based approach …

1. … comes with a natural representation of **empty nested tables**, and
2. … allows to “share” surrogates if **nested tables repeat.**

Add the following two rows to the NF² shapes table and consider the consequences (note: existing shape square and new shape rect use identical drawing commands):

<table>
<tr><th>id</th><th>shape</th><th>turtle</th></tr>
<tr><td>⋮</td><td>⋮</td><td>⋮</td></tr>
<tr><td>4</td><td>empty</td><td><table><tr><th>pos</th><th>command</th></tr><tr><td></td><td></td></tr></table></td></tr>
<tr><td>5</td><td>rect</td><td><table><tr><th>pos</th><th>command</th></tr><tr><td>1</td><td>(t,0,10)</td></tr><tr><td>2</td><td>(t,10,0)</td></tr><tr><td>3</td><td>(t,0,-10)</td></tr><tr><td>4</td><td>(t,-10,0)</td></tr></table></td></tr>
</table>

FROM NF² TO 1NF

- **Transforming data** from NF² to 1NF? **nf2to1nf**() ✓
- **Transforming queries** over NF² data to queries over 1NF data?

```
-- NF²: Find shapes drawn with multiple strokes
SELECT s.id, s.shape
FROM   shapes AS s
WHERE  EXISTS (SELECT 1
               FROM   s.turtle AS c              -- s.turtle has type table(…)
               WHERE  NOT (c.command).down);
```

```
-- 1NF: Find shapes drawn with multiple strokes
SELECT s.id, s.shape
FROM   shapes AS s
WHERE  EXISTS (SELECT 1
               FROM   (SELECT t.*                          } translation of
                        FROM   turtles AS t                } expression
                        WHERE  t.turtle = s.turtle) AS c   } s.turtle
               WHERE   NOT (c.command).down);
```

FROM NF² TO 1NF

Simulate a NF² RDBMS

- NF² to 1NF query transformation can be approached systematically as well. If we can transform data *and* queries automatically, we can **simulate a NF²-model RDBMS using a regular 1NF RDBMS.** (Hot research topic of the early 1990s.)

1. Accept **table and schema definitions** with table-valued columns.
   Behind the scenes: apply **nf2to1nf**() to generate equivalent 1NF table bundles.

2. Accept **DML** statements that insert (delete) table-valued column values.
   Behind the scenes: split inserted row into atomic/table-valued column values, distribute inserts between the 1NF tables of the bundle.

3. Accept **NF² SQL** queries that include functions over tables of values *xs*, e.g., EMPTY(*xs*), LENGTH(*xs*), *xs*[*i*], FORALL *x* IN *xs*: *p*(*x*), EXISTS *x* IN *xs*: *p*(*x*), …
   Behind the scenes: rewrite into regular SQL constructs that operate over the tables of the bundle.

FROM NF² TO 1NF

Simulate a NF² RDBMS

- Sample "NF² SQL" queries (⚑ marks NF² language constructs we have invented). Rewrite into regular SQL queries over 1NF table bundle shapes, turtles (see above).

```
-- What are the shapes with an empty drawing command list?
SELECT s.id, s.shape
FROM   shapes AS s
WHERE  ⚑EMPTY(s.turtle);

-- Which shapes are drawn with the pen down all the time?
SELECT s.id, s.shape
FROM   shapes AS s
WHERE  ⚑FORALL c IN s.turtle: (c.command).down

-- Which shapes contain strokes longer than 10 units?
SELECT s.id, s.shape
FROM   shapes AS s
WHERE  ⚑EXISTS c IN s.turtle: sqrt((c.command).x² + (c.command).y²) > 10
```

FROM NF² TO 1NF

Simulate a NF² RDBMS

- More sample “NF² SQL” queries:

```
-- First drawing command for each shape
SELECT s.id, s.shape, ⚑s.turtle[1].command AS head
FROM   shapes AS s;

-- Length of drawing command list for each shape ⚠️
SELECT s.id, s.shape, ⚑LENGTH(s.turtle)
FROM   shapes AS s;
```

- Most of these have a variety of translations to plain SQL (e.g., consider correlated subqueries vs. joins).
- ⚠️ Watch out for edge cases, in particular empty nested tables (see shape empty in table shapes)!