

# **INTRODUCTION TO RELATIONAL DATABASE SYSTEMS**

## **DATENBANKSYSTEME 1 (INF 3131)**

**Torsten Grust**  
**Universität Tübingen**  
Winter 2017/18

# DATABASE DESIGN

- Given a particular mini-world, almost always there will 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 said to be in **First Normal Form (1NF)**.

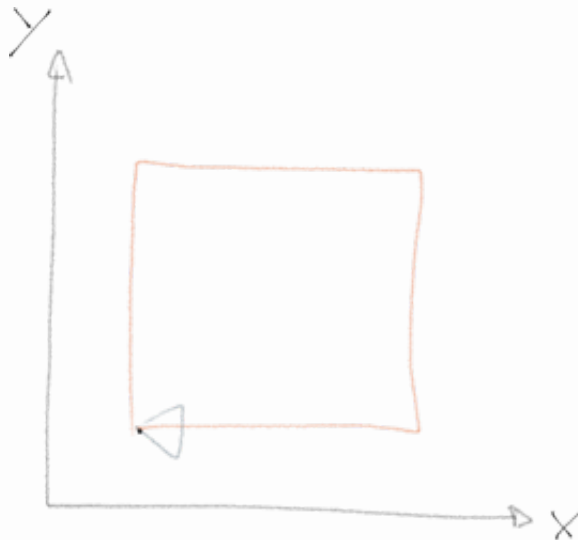
- Under this definition ...
  1. ... is a string (e.g., **text**) value in a table cell atomic?
  2. ... is a value of type **date** (with day, month, year components) atomic?
  3. ... is a value of a row type atomic?
  4. ... is an array of type  $t[]$  (with type  $t$  being atomic) atomic?
  5. ...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

<u>id</u>	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: <http://www.postgresql.org/docs/10/static/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 '`<p>,<x>,<y>;`' triples 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,  $i \geq 1$  (NULL if outside bounds)  
xs[<i>:<j>] -- array slice
```

- **Array operations:**

```
=, <>, <, > -- array to array comparison  
<expression> {=|<|>|...} {ANY|ALL}(xs) -- element to array comparison  
&gt;, <&, && -- 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`

<code>id</code>	<code>shape</code>	<code>turtle</code>
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[v1, v2, v3, ...])` yields

V<sub>1</sub>  
V<sub>2</sub>  
V<sub>3</sub>  
⋮

# 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<sup>2</sup>)**.

shapes

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



# TABLES IN TABLE CELLS (NF<sup>2</sup>)

- Notes:

1. Explicit `pos` column encodes command order (list semantics) in the nested tables.
2. Outer table `shape` has 3 rows. Type of `turtle` is `table(pos int, command cmd)`.
3. NF<sup>2</sup> admits recursion to arbitrary depth. “NF<sup>2</sup> SQL” queries reflect this recursion:

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

4. **⚠ No off-the-shelf RDBMS supports the NF<sup>2</sup> model** (mostly a 1980s research idea). Still a powerful/modular way to think about data modelling.
- Possible: Systematic (algorithmic) conversion of any NF<sup>2</sup> table into (a bundle of) equivalent 1NF tables.

# FROM NF<sup>2</sup> TO 1NF

`nf2to1nf( $R$ )` (input: table  $R$ , output: a table bundle of size  $\geq 1$ ):

- For each  $a \in \text{sch}(R)$  do
  - If  $\text{type}(a) = \text{table}(\underline{b_1 t_1, \dots, b_k t_k}, \dots, b_m t_m)$  then
    1. Create a new table  $R_a(\underline{a \text{ surrogate}, b_1 t_1, \dots, b_k t_k}, \dots, b_m t_m)$
    2. For each row  $r \in \text{inst}(R)$  do
      1. Create a new value  $\tau$  of type **surrogate**
      2. If table  $r.a$  is not empty then
        - For each row  $(v_1, \dots, v_m)$  in  $r.a$  do
          - Insert row  $(\tau, v_1, \dots, v_m)$  into  $R_a$
      3. Set  $r.a$  to  $\tau$  [if  $r.a$  is empty,  $\tau$  will not have a match in  $R_a$ ]
    3. Set  $\text{type}(a)$  to **surrogate**
    4. `nf2to1nf( $R_a$ )`

# FROM NF<sup>2</sup> TO 1NF

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

`shapes` ( $R$ )

<u>id</u>	shape	turtle
1	square	$\tau_1$
2	triangle	$\tau_2$
3	cross	$\tau_3$

`turtles` ( $R_{\text{turtle}}$ )

<u>turtle</u>	<u>pos</u>	command
$\tau_1$	1	(t,0,10)
$\tau_1$	2	(t,10,0)
$\tau_1$	3	(t,0,-10)
$\tau_1$	4	(t,-10,0)
$\tau_2$	1	(t,5,10)
$\tau_2$	2	(t,5,-10)
$\tau_2$	3	(t,-10,0)
$\tau_3$	1	(t,0,10)
$\tau_3$	2	(f,-5,-5)
$\tau_3$	3	(t,10,0)

# FROM NF<sup>2</sup> 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 NF<sup>2</sup> **shapes** table and consider the consequences (note: existing shape **square** and new shape **rect** use identical drawing commands):

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

# FROM NF<sup>2</sup> TO 1NF

- **Transforming data** from NF<sup>2</sup> to 1NF? `nf2to1nf()` ✓
- **Transforming queries** over NF<sup>2</sup> data to queries over 1NF data?

```
-- NF2: Find shapes drawn with multiple strokes
SELECT s.id, s.shape
FROM   shapes s
WHERE  EXISTS (SELECT 1
               FROM   s.turtle 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 s
WHERE  EXISTS (SELECT 1
               FROM   (SELECT t.*          } translation of
                       FROM   turtles t   } expression
                       WHERE  t.turtle = s.turtle) c } s.turtle
               WHERE  NOT (c.command).down);
```

# FROM NF<sup>2</sup> TO 1NF

## Simulate a NF<sup>2</sup> RDBMS

- NF<sup>2</sup> to 1NF query transformation can be approached systematically as well. If we can transform data *and* queries automatically, we can **simulate a NF<sup>2</sup>-model RDBMS using a regular 1NF RDBMS**. (Hot research topic of the early 1990s.)
  1. Accept **table and schema definitions** with table-valued (or list-valued) columns.  
*Behind the scenes:* apply `nf2to1nf()` to generate equivalent 1NF table bundles.
  2. Accept **DML** statements that also 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 **NF<sup>2</sup> SQL** queries that include functions over tables (lists) 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 NF<sup>2</sup> TO 1NF

## Simulate a NF<sup>2</sup> RDBMS

- Sample “NF<sup>2</sup> SQL” queries (♣ marks NF<sup>2</sup> 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 s  
WHERE  ♣EMPTY(s.turtle);
```

```
-- Which shapes are drawn with the pen down all the time?  
SELECT s.id, s.shape  
FROM   shapes 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 s  
WHERE  ♣EXISTS c IN s.turtle: sqrt((c.command).x2 + (c.command).y2) > 10
```

# FROM NF<sup>2</sup> TO 1NF

## Simulate a NF<sup>2</sup> RDBMS

- More sample “NF<sup>2</sup> SQL” queries:

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

```
-- Length of drawing command list for each shape ⚠
SELECT s.id, s.shape, LENGTH(s.turtle)
FROM   shapes 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](#))!