INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2021/22

CONSTRAINTS

- Recall the recent correlated SQL query returning the LEGO bricks in any of the animal-related categories:

```
SELECT b.name
FROM   bricks AS b
WHERE  (SELECT c.name
        FROM   categories AS c
        WHERE  b.cat = c.cat) ~ 'Animal'
```

  - In the subquery, we assume that there
    - ❶ **exists a row** in categories whose cat identifier matches that of brick b, and there is
    - ❷ **no more than one row** of categories with a matching cat identifier.
- A violation of these assumptions means that the database state is *not* a valid image of the mini-world. Clearly, a job for **constraints**.
- A formulation of the required constraint **spans two tables** (inter-table constraint between **source** bricks and **target** categories).

VALUE-BASED REFERENCES

bricks

piece	type	name	cat	weight	img	x	y	z
			c_0					

- **Violation of assumption 1** ($c_0 \notin \{c_1, \ldots, c_n\}$, no match in target column):

categories

cat	name
c_1	
$\vdots$	
c_n	

- **Violation of assumption 2** (more than one match in target column):

categories

cat	name
c_0	
$\vdots$	
c_0	

VALUE-BASED REFERENCES

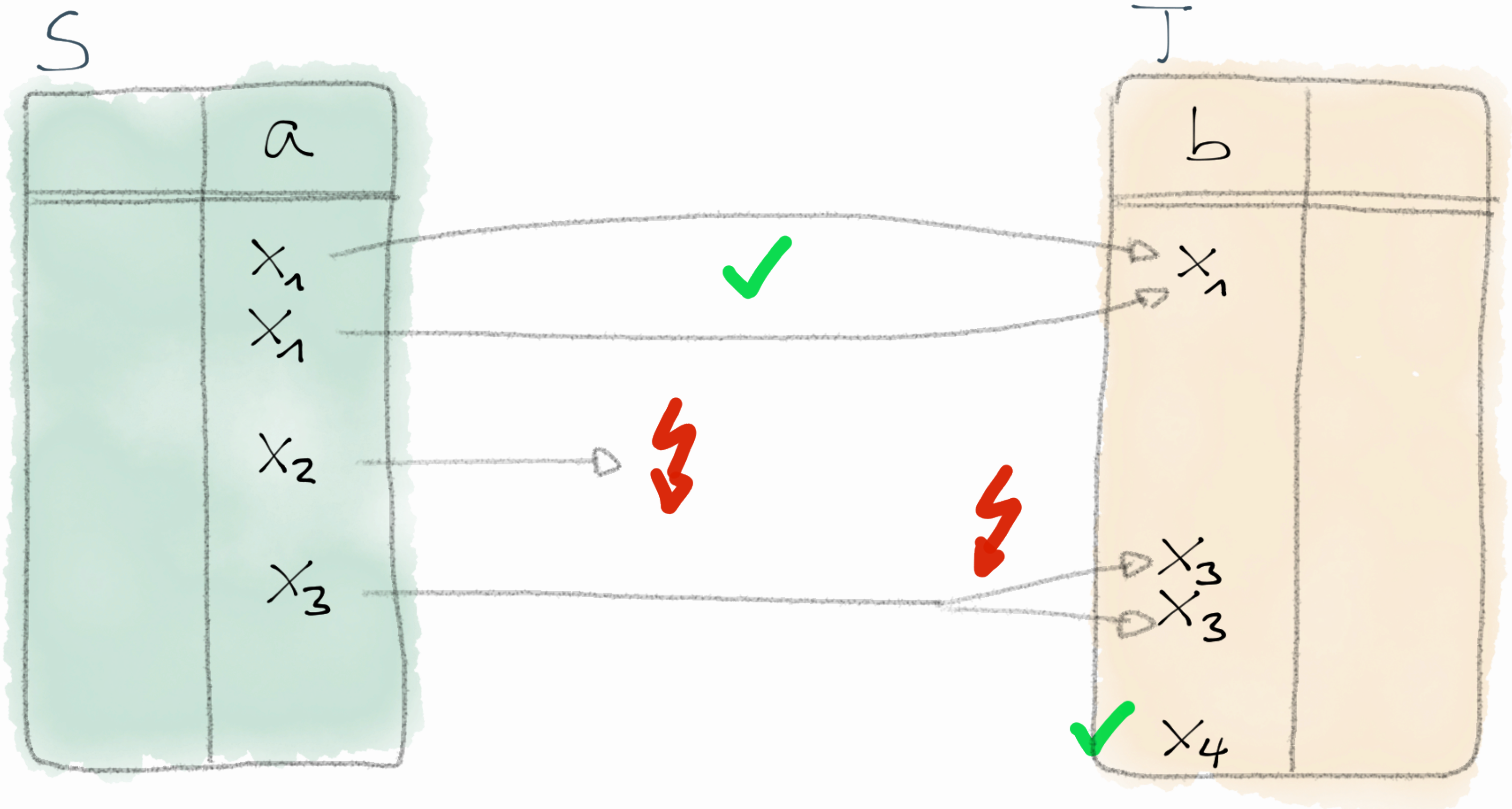
- If both assumptions hold, we may safely use **value equality** to **implement references between rows** in separate tables/in the same table.
  - Recall the flat representation of the LEGO mini-world using tables contains, bricks, minifigs in our discussion of data models.

Pointers vs. Value-Based References

Pointer	Value-based Reference
points to address of object *o*	contains value that uniquely identifies target row *o*
is dangling	contains value not found in target column ⚠
–	contains value that is not unique in target column ⚠
is dereferenced	query target table for *the* row containing the value

- In SQL, a **join** between source and target table **dereferences multiple value-based references at once.**

VALUE-BASED REFERENCES



Value-based references between source table *S* and target table *T*.

FOREIGN KEYS

Foreign Key Constraint

Let (S, α) and (T, β) denote two relational schemata (not necessarily distinct), where $K = \{b_1,\dots,b_k\} \subseteq \beta$ is a key of T. Let $F = \{a_1,\dots,a_k\} \subseteq \alpha$ with $\mathit{type}(a_i) = \mathit{type}(b_i)$, $i = 1,\dots,k$.

F is a **foreign key in** S **referencing** T, if

$$\forall\ s \in \mathit{inst}(S):\ \exists\ t \in \mathit{inst}(T):\ s.F = t.K$$

- Notes:
  - The ∀∃ condition validates assumption ❶. K being a key in target T validates assumption ❷.
  - ⚠️ In general, **foreign key** F **is not a key** in source table S: two rows s_1, $s_2 \in \mathit{inst}(S)$ with $s_1.F = s_2.F$ can refer to the same row in target T.

FOREIGN KEYS: REFERENTIAL INTEGRITY

- Foreign key constrains also go under the name of **inclusion constraints**, since we have

```
SELECT s.a₁, …, s.aₖ          □          SELECT t.b₁, …, t.bₖ
FROM   S AS s                            FROM   T AS t
```

  (Quiz: Insert □ ∈ { ⊆, =, ⊇ } above.)

- If we declare the foreign key constraint with ALTER TABLE, the RDBMS refuses any database state change that violates the above inclusion and thus the **referential integrity** of the database.
  If a row's foreign key value contains NULL, that row is excluded from the integrity check.

- Referential integrity may be lost whenever

1. rows are **inserted into source** table *S* or
2. rows are **deleted from/updated in target** table *T*.

SQL: FOREIGN KEYS

ALTER TABLE … FOREIGN KEY … REFERENCES

The SQL DDL command

```
ALTER TABLE [ IF EXISTS ] source
  ADD FOREIGN KEY (column_name [, …]) REFERENCES target
  [ ON DELETE action ] [ ON UPDATE action ]
```

establishes a **foreign key** in *source* referencing (the primary key of) *target*. If referenced target rows are deleted/updated, perform *action*:

```
-- default: if referential integrity is lost: do not update, yield error
  NO ACTION
-- delete/update any source row referencing the deleted/update target row
  CASCADE
-- set foreign key to NULL in the source rows referencing the target row
  SET NULL
```

SQL: QUANTIFICATION

EXISTS / IN

The SQL predicate

```
[NOT] EXISTS(query)
```

yields true [false] if *query* returns **one row or more.** The SQL predicate

```
expression [NOT] IN (query)
```

checks whether **any [no] value** returned by *query* **equals** *expression*.

- These predicates provide a form of **existential** and **universal quantification** in SQL:
  expression IN (*query*) $\equiv \exists\ r \in query\!: r = expression$
  expression NOT IN (*query*) $\equiv \forall\ r \in query\!: r \neq expression$

SQL: REFERENTIAL INTEGRITY

- With EXISTS and IN we can formulate referential integrity and check the inclusion constraint in SQL itself:
- Detect if inclusion constraint is violated:

```
SELECT s.a₁, …, s.aₖ          ⊈          SELECT t.b₁, …, t.bₖ
FROM   S AS s                            FROM   T AS t
```

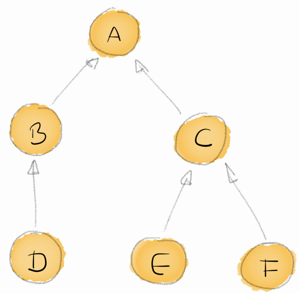
Equivalent formulation in SQL (SQL does not define operators $\subseteq$ or $\nsubseteq$):

```
EXISTS(SELECT 1
       FROM   S AS s
       WHERE  ROW(s.a₁, …, s.aₖ) NOT IN (SELECT t.b₁, …, t.bₖ
                                         FROM   T AS t))
```

- ⚠️ Note that expression 1 in the outer SELECT clause is indeed arbitrary (any expression will do).

INTRA-TABLE FOREIGN KEYS

- Foreign keys help to relate the rows of a source table *S* and a target table *T*. But *S* and *T* need *not* be different. We end up with **intra-table references.**
- **Example:** Representation of **tree-shaped data structures** using a table (foreign key parent references key node):



tree

<u>node</u>	parent
A	□
B	A
C	A
D	B
E	C
F	C

INTRA-TABLE FOREIGN KEYS: QUERIES

- Queries over such self-referencing tables often lead to **self-joins** in which the rows of a table are related to (other) rows of the same table.
- Consider:

```sql
-- What are the labels of the siblings of the node with label E?
SELECT t2.node
FROM   tree AS t1, tree AS t2
WHERE  t1.node = 'E'
AND    t1.parent = t2.parent
```

```sql
-- What are the labels of the grandchildren of the node with label A?
SELECT t3.node
FROM   tree AS t1, tree AS t2, tree AS t3
WHERE  t1.node = 'A'
AND    t2.parent = t1.node
AND    t3.parent = t2.node
```

INTRA-TABLE FOREIGN KEYS: UPDATES

- The population of self-referencing tables requires some care since referential integrity must not be violated at any point in time.
- Possible strategies:

1. Insert in **topological order:** Insert root(s) of data structure first, since their foreign keys will be NULL (here: node A), then proceed with the roots of the sub-structures.
   If this is no option (cyclic structure):
2. Use **bulk insert:** insert *all* rows of table using a *single* SQL DML statement (e.g. INSERT INTO). Referential integrity is checked *after* statement completion.
3. Insert referencing rows with **NULL foreign key.** Then insert referenced rows. Finally, use UPDATE … SET … to establish the correct foreign key value in referencing rows.
4. **Temporarily disable referential integrity** checking, populate table in any row order, re-enable referential integrity.

CONSTRAINTS — SUMMARY

- The **constraint set** $\mathbb{C}$ is integral part of a relational database schema:

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

- Any valid database state has to satisfy all integrity constraints (= predicates) of $\mathbb{C}$.
- Benefits of constraints:
  - **Protection** against (many) **data input errors.**
  - **Formal documentation** of the database schema.
  - Automatic **enforcement of law/company standards.**
  - **Protection against inconsistency** if data is stored redundantly.
  - Queries/application **programs become simpler if** developers may assume that retrieved **data fulfills certain properties.**