INTRODUCTION TO
RELATIONAL DATABASE SYSTEMS
DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2017/18
CONSTRAINTS

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

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

- In the subquery, we assume that there

1. exists a row in categories whose cat identifier matches that of brick b, and
2. is 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

<table>
<thead>
<tr>
<th>bricks</th>
<th>piece</th>
<th>type</th>
<th>name</th>
<th>cat</th>
<th>weight</th>
<th>img</th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>c</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Violation of assumption #1** ($\forall_i c_i \neq c$, no match in target column):

  \[
  c_1 \\
  \vdots \\
  c_n \\
  c
  \]

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

  \[
  c \\
  \vdots \\
  c
  \]
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

<table>
<thead>
<tr>
<th>Pointer</th>
<th>Value-based Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>points to address of object $o$</td>
<td>contains value that uniquely identifies target row $o$</td>
</tr>
<tr>
<td>is dangling</td>
<td>contains value not found in target column $A$</td>
</tr>
<tr>
<td>is dereferenced</td>
<td>contains value that is not unique in target column $A$</td>
</tr>
<tr>
<td>query target table for the row containing the value</td>
<td></td>
</tr>
</tbody>
</table>

- In SQL, a join between source and target table dereferences multiple value-based references at once.
Value-based references between source table $S$ and target table $T$. 
FOREIGN KEYS

Foreign Key Constraint

Let \((S, \alpha)\) and \((T, \beta)\) denote two relational schemata (not necessarily distinct), where \(K = \{b_{j_1}, \ldots, b_{j_k}\} \subseteq \beta\) is the primary key of \(T\). Let \(F = \{a_{i_1}, \ldots, a_{i_k}\} \subseteq \alpha\) with \(\text{type}(a_{i_h}) = \text{type}(b_{j_h}), h = 1, \ldots, k\).

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

\[\forall s \in \text{inst}(S) : \exists t \in \text{inst}(T) : s.F = t.K\]

- **Notes:**
  - The \(\forall \exists\) condition validates assumption #1. \(K\) being a key in target \(T\) validates assumption #2.
  - ⚠ In general, **foreign key \(F\) is not a key** in source table \(S\). Two rows \(s_1, s_2 \in \text{inst}(S)\) with \(s_1.F = s_2.F\) can refer to the same row in target \(T\).
FOREIGN KEYS: REFERENTIAL INTEGRITY

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

\[
\begin{align*}
\text{SELECT } & s.\langle a_{i1} \rangle, \ldots, s.\langle a_{ik} \rangle & \quad \Box \\
\text{FROM } & <S> \ s
\end{align*}
\]

\[
\begin{align*}
\text{SELECT } & t.\langle b_{j1} \rangle, \ldots, t.\langle b_{jk} \rangle \\
\text{FROM } & <T> \ t
\end{align*}
\]

(Quiz: Insert \( \Box \in \{ \subseteq, =, \supseteq \} \) 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 \( T \).
ALTER TABLE ... FOREIGN KEY ... REFERENCES

The SQL DDL command

ALTER TABLE [ IF EXISTS ] <source>
  ... target row
  CASCADE
-- set foreign key to NULL in the source rows referencing the target row
  SET NULL

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
The SQL predicate 

\[ \text{NOT EXISTS}(<\text{query}>) \]

yields true [false] if \(<\text{query}\) returns one row or more. The SQL predicate

\[ <\text{expression}> \text{ NOT IN } ( <\text{query}> ) \]

checks whether any [no] value returned by \(<\text{query}>\) equals \(<\text{expression}>\).

- These predicates provide a form of existential and universal quantification in SQL:
  \[ <\text{expression}> \text{ IN } ( <\text{query}> ) \equiv \exists r \in <\text{query}>: r = <\text{expression}> \]
  \[ <\text{expression}> \text{ NOT IN } ( <\text{query}> ) \equiv \forall r \in <\text{query}>: r \neq <\text{expression}> \]
- With **EXISTS** and **IN** we can formulate referential integrity and check the inclusion constraint in SQL itself:

- Detect if inclusion constraint is violated:

```sql
SELECT s.<a₁₁>, ..., s.<a₁_k>  \∉  SELECT t.<b₁₁>, ..., t.<b₁_k>
FROM  <S> s
FROM  <T> t
```

Equivalent formulation in SQL (SQL does not define a \(\subseteq\) or \(\not\subseteq\)):

```sql
EXISTS(SELECT 1
      FROM  <S> s
      WHERE  ROW(s.<a₁₁>, ..., s.<a₁_k>) NOT IN (SELECT t.<b₁₁>, ..., t.<b₁_k>
                                                  FROM  <T> 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**):

```
<table>
<thead>
<tr>
<th>node</th>
<th>parent</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>□</td>
</tr>
<tr>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>C</td>
<td>A</td>
</tr>
<tr>
<td>D</td>
<td>B</td>
</tr>
<tr>
<td>E</td>
<td>C</td>
</tr>
<tr>
<td>F</td>
<td>C</td>
</tr>
</tbody>
</table>
```

Tree:
```
A
  /\  
B   C
  / \  /
D   E C
  \  /  \
  F
```
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:

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

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

- The constraint set $\mathcal{C}$ is integral part of a relational database schema:
  $$((R_1, \alpha_1), (R_2, \alpha_2), \ldots, \mathcal{C})$$

- Any valid database state has to satisfy all integrity constraints (= predicates) of $\mathcal{C}$.

- 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.