INTRODUCTION TO
RELATIONAL DATABASE SYSTEMS
DATENBANKSYSTEME 1 (INF 3131)

Torsten Grust
Universität Tübingen
Winter 2017/18
LEGGO BUILDING INSTRUCTIONS

- Each LEGO set comes with building instructions, an illustrated booklet that details the individual steps of model construction.
- One page in the booklet holds one or more instruction steps (steps are numbered 1, 2, ...).
- Each step lists the pieces (with their color and quantity) required to complete the step.
- Each step comes with an illustration of where the listed pieces find their place in the model.

- What would be a reasonable design for a building instructions table? Clearly:
  1. Do not include LEGO set details in instructions: instead, use a foreign key to refer to table sets.
  2. Do not include LEGO piece details in instructions: instead, use a foreign key to refer to table bricks.
  3. Represent page numbers, step numbers, image sizes as integers but formulate constraints that avoid data entry errors (e.g. negative page/step numbers).
## LEGO BUILDING INSTRUCTIONS (TABLE DESIGN)

### Instructions

<table>
<thead>
<tr>
<th>set</th>
<th>step</th>
<th>piece</th>
<th>color</th>
<th>quantity</th>
<th>page</th>
<th>img</th>
<th>width</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>9495–1</td>
<td>7</td>
<td>3010</td>
<td>2</td>
<td>2</td>
<td>24</td>
<td>image07</td>
<td>639</td>
<td>533</td>
</tr>
<tr>
<td>9495–1</td>
<td>7</td>
<td>3023</td>
<td>2</td>
<td>2</td>
<td>24</td>
<td>image07</td>
<td>639</td>
<td>533</td>
</tr>
<tr>
<td>9495–1</td>
<td>7</td>
<td>2877</td>
<td>86</td>
<td>1</td>
<td>24</td>
<td>image07</td>
<td>639</td>
<td>533</td>
</tr>
<tr>
<td>9495–1</td>
<td>8</td>
<td>3002</td>
<td>7</td>
<td>2</td>
<td>24</td>
<td>image08</td>
<td>650</td>
<td>522</td>
</tr>
<tr>
<td>9495–1</td>
<td>8</td>
<td>30414</td>
<td>1</td>
<td>2</td>
<td>24</td>
<td>image08</td>
<td>650</td>
<td>522</td>
</tr>
<tr>
<td>9495–1</td>
<td>9</td>
<td>30414</td>
<td>85</td>
<td>1</td>
<td>25</td>
<td>image09</td>
<td>541</td>
<td>638</td>
</tr>
<tr>
<td>9495–1</td>
<td>9</td>
<td>3062b</td>
<td>85</td>
<td>2</td>
<td>25</td>
<td>image09</td>
<td>541</td>
<td>638</td>
</tr>
<tr>
<td>9495–1</td>
<td>10</td>
<td>30033</td>
<td>11</td>
<td>1</td>
<td>25</td>
<td>image10</td>
<td>540</td>
<td>662</td>
</tr>
<tr>
<td>9495–1</td>
<td>10</td>
<td>2412b</td>
<td>86</td>
<td>1</td>
<td>25</td>
<td>image10</td>
<td>540</td>
<td>662</td>
</tr>
<tr>
<td>9495–1</td>
<td>10</td>
<td>4589b</td>
<td>86</td>
<td>2</td>
<td>25</td>
<td>image10</td>
<td>540</td>
<td>662</td>
</tr>
<tr>
<td>9495–1</td>
<td>10</td>
<td>87580</td>
<td>85</td>
<td>1</td>
<td>25</td>
<td>image10</td>
<td>540</td>
<td>662</td>
</tr>
<tr>
<td>9495–1</td>
<td>11</td>
<td>3039</td>
<td>2</td>
<td>1</td>
<td>25</td>
<td>image11</td>
<td>1042</td>
<td>558</td>
</tr>
<tr>
<td>9495–1</td>
<td>11</td>
<td>4073</td>
<td>85</td>
<td>4</td>
<td>25</td>
<td>image11</td>
<td>1042</td>
<td>558</td>
</tr>
<tr>
<td>9495–1</td>
<td>11</td>
<td>44728</td>
<td>3</td>
<td>1</td>
<td>25</td>
<td>image11</td>
<td>1042</td>
<td>558</td>
</tr>
</tbody>
</table>
REDUNDANCY

- The design of table instructions appears reasonable. We immediately spot a fair amount of redundancy, though. For example:

1. Step 10 of Set 9495 is printed on page 25. [represented 4 ×]
2. Step 7 of Set 9495 is illustrated by `<image07>`. [3 ×]
3. `<image09>` has dimensions 541 × 638 pixels. [2 ×]

- **Redundancy** comes with a number of serious problems, most importantly:
  
  - **Storage space is wasted.**
    Tables occupy more disk space than needed. Query processor has to touch/move more bytes. Archival storage (backup) requires more resources.
  
  - **Redundant copies will go out of sync.**
    Eventually, an update operation will miss a copy. The database instance now contains “multiple truths.” Typically, this goes unnoticed by DBMS and user.
EMBEDDED FUNCTIONS AND REDUNDANCY

- In table instructions, the source of redundancy is the presence of functions that are embedded in the table.

Leibniz Principle
If \( f \) is a function defined on \( x, y \), then

\[
x = y \land f(x) = z \Rightarrow f(y) = z
\]

- Table instructions embeds the materialized functions
1. printed_on(): maps set, step to the page it is printed on
2. illustrated_by(): maps set, step to the illustration stored in image img
3. image_size(): maps an image img to its width and height
FUNCTIONAL DEPENDENCIES

Functional Dependency (FD)

Let \((R, \alpha)\) denote a relational schema. Given \(\beta \subseteq \alpha\) and \(c \in \alpha\), the functional dependency \(\beta \rightarrow c\) holds in \(R\) if

\[
\forall t, u \in \text{inst}(R) : t.\beta = u.\beta \Rightarrow t.c = u.c
\]

Read: “If two rows agree on the columns in \(\beta\), they also agree on column \(c\).” (\(\beta\): function arguments, \(c\): function result).

Notation: the FD \(\beta \rightarrow \{c_1, \ldots, c_n\}\) abbreviates the set of FDs \(\beta \rightarrow c_1, \ldots, \beta \rightarrow c_n\).

- Note: If \(c \in \beta\), then \(\beta \rightarrow c\) is called a trivial FD that obviously holds for any instance of \(R\). No interesting insight into \(R\) here.
FUNCTIONAL DEPENDENCIES

- FDs are constraints that document universally valid mini-world facts (e.g., “a step is associated with one illustration”). FDs thus need to hold in all database instances.

### instructions

<table>
<thead>
<tr>
<th>set</th>
<th>step</th>
<th>piece</th>
<th>color</th>
<th>quantity</th>
<th>page</th>
<th>img</th>
<th>width</th>
<th>height</th>
</tr>
</thead>
<tbody>
<tr>
<td>9495-1</td>
<td>7 3010</td>
<td>2</td>
<td>2</td>
<td>24 &lt;image07&gt;</td>
<td>639</td>
<td>533</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9495-1</td>
<td>7 3023</td>
<td>2</td>
<td>2</td>
<td>24 &lt;image07&gt;</td>
<td>639</td>
<td>533</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9495-1</td>
<td>7 2877</td>
<td>86</td>
<td>1</td>
<td>24 &lt;image07&gt;</td>
<td>639</td>
<td>533</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9495-1</td>
<td>8 3002</td>
<td>7</td>
<td>2</td>
<td>24 &lt;image08&gt;</td>
<td>650</td>
<td>522</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9495-1</td>
<td>8 30414</td>
<td>1</td>
<td>2</td>
<td>24 &lt;image08&gt;</td>
<td>650</td>
<td>522</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9495-1</td>
<td>9 30414</td>
<td>85</td>
<td>1</td>
<td>25 &lt;image09&gt;</td>
<td>541</td>
<td>638</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9495-1</td>
<td>9 3062b</td>
<td>85</td>
<td>2</td>
<td>25 &lt;image09&gt;</td>
<td>541</td>
<td>638</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Which functional dependencies hold in table `instructions`?
FUNCTIONAL DEPENDENCIES

- Given table $R$, check whether the FD $\{b_1, \ldots, b_n\} \rightarrow c$ holds in the current table instance:

$$\text{SELECT DISTINCT 'The FD } \{b_1, \ldots, b_n\} \rightarrow c \text{ does not hold' FROM } \ R \ \text{GROUP BY } b_1, \ldots, b_n \ \text{HAVING COUNT(DISTINCT c) > 1}$$

Aggregate Functions

Optional modifier DISTINCT affects the computation of aggregate functions:

- $\langle\text{aggregate}\rangle([\text{ ALL }] \langle\text{expression}\rangle)$ -- aggregate all non-NULL values
- $\langle\text{aggregate}\rangle(\text{DISTINCT} \langle\text{expression}\rangle)$ -- aggregate all distinct non-NULL values
- $\langle\text{aggregate}\rangle(*)$ -- aggregate all rows (count(*))
Note that a **key** implicitly defines a particularly **strong FD**: the key columns functionally determine all columns of the table.

**Keys vs FDs (1)**
Assume table \((R, \{a_1, \ldots, a_k, a_{k+1}, \ldots, a_n\})\).

\[
\{a_1, \ldots, a_k\} \text{ is a key of } R \iff \{a_1, \ldots, a_k\} \rightarrow \{a_{k+1}, \ldots, a_n\} \text{ holds.}
\]

- So, keys are special FDs.
- Turning this around: FDs are a generalization of keys.
FD → (LOCAL, PARTIAL) KEY

**Keys vs FDs (2)**

Assume table $R$ and FD $\beta \rightarrow c$. Then $\beta$ is key in the sub-table of $R$ defined by

$$\text{SELECT DISTINCT } \beta, c$$
$$\text{FROM } R$$

**Example:** for table `instructions` and FD $\{ \text{set, step} \} \rightarrow \text{page}$ the sub-table is

<table>
<thead>
<tr>
<th>set</th>
<th>step</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
<tr>
<td>9495-1</td>
<td>7</td>
<td>24</td>
</tr>
<tr>
<td>9495-1</td>
<td>8</td>
<td>24</td>
</tr>
<tr>
<td>9495-1</td>
<td>9</td>
<td>25</td>
</tr>
<tr>
<td>9495-1</td>
<td>10</td>
<td>25</td>
</tr>
<tr>
<td>9495-1</td>
<td>11</td>
<td>25</td>
</tr>
<tr>
<td>:</td>
<td>:</td>
<td>:</td>
</tr>
</tbody>
</table>

(i.e., exactly the table materializing the function `printed_on()`, see above).
- Example: recall table stores of the LEGO Data Warehouse scenario:

<table>
<thead>
<tr>
<th>store</th>
<th>city</th>
<th>state</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HAMBURG</td>
<td></td>
<td>Germany</td>
</tr>
<tr>
<td></td>
<td>LEIPZIG</td>
<td></td>
<td>Germany</td>
</tr>
<tr>
<td>8</td>
<td>MÜNCHEN</td>
<td></td>
<td>Germany</td>
</tr>
<tr>
<td>9</td>
<td>MÜNCHEN PASING</td>
<td></td>
<td>Germany</td>
</tr>
<tr>
<td>10</td>
<td>NÜRNBERG</td>
<td></td>
<td>Germany</td>
</tr>
<tr>
<td>16</td>
<td>ARDEN FAIR MALL</td>
<td></td>
<td>USA</td>
</tr>
<tr>
<td>17</td>
<td>DISNEYLAND RESORT</td>
<td></td>
<td>USA</td>
</tr>
<tr>
<td>18</td>
<td>FASHION VALLEY</td>
<td></td>
<td>USA</td>
</tr>
</tbody>
</table>

- List the FDs that hold in table stores.
- Does the mini-world suggest FDs not implied by the rows shown above?
FUNCTIONAL DEPENDENCIES

- An FD indicates the presence of a materialized function. Consider the following variant of the users and ratings table:

<table>
<thead>
<tr>
<th>user</th>
<th>rating</th>
<th>stars</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>3</td>
<td>***</td>
</tr>
<tr>
<td>Bert</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>Cora</td>
<td>4</td>
<td>****</td>
</tr>
<tr>
<td>Drew</td>
<td>5</td>
<td>*****</td>
</tr>
<tr>
<td>Erik</td>
<td>1</td>
<td>*</td>
</tr>
<tr>
<td>Fred</td>
<td>3</td>
<td>***</td>
</tr>
</tbody>
</table>

- FD \{ rating \} \rightarrow stars materializes the computable function \( stars = f(\text{rating}) = \text{repeat}('\ast', \text{rating}) \) [see PostgreSQL’s string function library].

- In such cases, good database design should consider to trade materialization for computation. Removes redundancy.
CREATE VIEW

Binds `<query>` to `<name>` which is globally visible. Whenever table `<name>` is referenced in subsequent queries, `<query>` is **re-evaluated** and its result returned *(no materialization* of the result of `<query>` is performed):

```
-- TEMPORARY: automatically drop view after current session
CREATE [ OR REPLACE ] [ TEMPORARY ] VIEW `<name>`
  AS `<query>`
```

- Compare with CTEs: local visibility in surrounding **WITH** statement only.
- A temporary view named `<name>` shadows a (regular, persistent) table of the same name.
- Views provide **data independence**: users and applications continue to refer to `<name>`, while the database designer may decide to replace a persistent table with a computed query or vice versa.

- **Example**: turn the materialized function $stars = f(rating)$ into a computed function:

  ```sql
  -- drop the materialized function from the table
  ALTER TABLE users
      DROP COLUMN stars;

  -- provide the three-column table that users/applications expect
  CREATE TEMPORARY VIEW users(user, rating, stars)
      AS SELECT u.user, u.rating, repeat('*', u.rating) stars
      FROM    users u;
  ```

- Since PostgreSQL's `repeat()` is a pure function, the FD $rating \rightarrow stars$ trivially holds in the view.
DERIVING FUNCTIONAL DEPENDENCIES

- Given a set \( F \) of FDs over table \( R \), simple inference rules—the **Armstrong Axioms**—suffice to generate all FDs following from those in \( F \).

**Armstrong Axioms**

Apply exhaustively to generate all FDs implied by FD set \( F \).

**Reflexivity:**

If \( \gamma \subseteq \beta \), then \( \beta \rightarrow \gamma \).

**Augmentation (with \( \beta \cup \{c\} \rightarrow \gamma \cup \{c\} \))**:

If \( \beta \rightarrow \gamma \), then \( \beta \cup \{c\} \rightarrow \gamma \cup \{c\} \)

**Transitivity:**

If \( \alpha \rightarrow \beta \) and \( \beta \rightarrow \gamma \), then \( \alpha \rightarrow \gamma \).

- Note: transitivity closely relates to **function composition**: if \( f, g \) are functions, so is \( g \circ f \).
DERIVING FDS (COVER)

- **Problem:** Given a set $\alpha \subseteq \text{sch}(R)$ of columns and a set of FDs $F$ over $R$, compute the **cover** $\alpha^+$, i.e. the set of all columns functionally determined by $\alpha$.

**Cover**

The **cover** $\alpha^+$ of a set of columns $\alpha$ is the set of all columns $c$ that are functionally determined by the columns in $\alpha$ (with respect to a given FD set $F$):

$$\alpha^+ := \{c \mid F \text{ implies } \alpha \rightarrow c\}$$

- ⚠ Should we find that $\alpha^+ = \text{sch}(R)$, then $\alpha$ is a candidate key for $R$. 
DERIVING FDS (COVER)

- Compute the cover $\alpha^+$ for a given set of FDs:

\[
\text{cover}(\alpha, F) \quad (\text{Input: column set } \alpha, \text{ FD set } F, \text{ Output: } \alpha^+) 
\]

1. $X := \alpha$
2. Repeat
   - For each FD $\beta \rightarrow c$ in $F$ do
     - If $\beta \subseteq X$ then
       - $X := X \cup \{c\}$
   Until $X$ did not change
3. Return $X$
- **Example:** In table instructions, compute \{set, step\}⁺ with \( F = \{ \{ \text{set, step} \} \rightarrow \text{page}, \{ \text{set, step} \} \rightarrow \text{img}, \{ \text{img} \} \rightarrow \text{width}, \{ \text{img} \} \rightarrow \text{height} \} \).

\[
\begin{array}{cccccccc}
\text{instructions} & \text{set} & \text{step} & \text{piece} & \text{color} & \text{quantity} & \text{page} & \text{img} & \text{width} & \text{height} \\
\end{array}
\]

- Tracing column set \( X \):
  1. \( X := \{ \text{set, step} \} \)
  2. FD \{set, step\} \rightarrow \text{page}, \{set, step\} \subseteq X: X := X \cup \{ \text{page} \}  
    FD \{set, step\} \rightarrow \text{img}, \{set, step\} \subseteq X: X := X \cup \{ \text{img} \}  
    FD \{\text{img}\} \rightarrow \text{width}, \{\text{img}\} \subseteq X: X := X \cup \{ \text{width} \}  
    FD \{\text{img}\} \rightarrow \text{height}, \{\text{img}\} \subseteq X: X := X \cup \{ \text{height} \}  

All FDs considered. \( X = \{ \text{set, step, page, img, width, height} \} \)

Repetition of 2. does not add new columns to \( X \).

3. Return \{set, step, page, img, width, height\}. 
DERIVING CANDIDATE KEYS

\text{key}(K, U, F) \ (\text{Input: FD set } F, \text{ Output: set of all candidate keys for } R)\]

- If $U = \emptyset$ then Return $\{K\}$ \hspace{1cm} [Invariant: \text{cover}(K \cup U, F) = \text{sch}(R)]
  
  else
  
  - $X := \emptyset$

  - For each $c \in U$ do

    - If $c \notin \text{cover}(K \cup (U \setminus \{c\}), F)$ then \hspace{1cm} [Is $c$ essential for the key?]

      - $X := X \cup \text{key}(K \cup \{c\}, U \setminus \{c\}, F)$

    else

      - $X := X \cup \text{key}(K, U \setminus \{c\}, F)$

  - Return $X$

- Invoke via $\text{key}(\emptyset, \text{sch}(R), F)$.

- Can optimize at $\triangledown$ : invoke $\text{key}(K \cup \{c\}, U \setminus \text{cover}(K \cup \{c\}, F), F)$ instead.
Typically it is a severe sign of **poor database design if tables embed functions**, i.e. if a table contains **FDs that are not implied by the primary key. ❗️**

- Consequences of table designs with non-key FDs / embedded functions:
  1. **Redundancy** (see above ✔️)
  2. Update/Insertion/Deletion **Anomalies**
  3. **RDBMS cannot protect the integrity** of non-key FDs, thus risk of inconsistency over time:
     - SQL DDL does *not* implement an `ALTER TABLE ... ADD FUNCTIONAL DEPENDENCY ...` statement.
     - Although FDs embody important mini-world facts they are easily violated without protection. (Can simulate this protection using SQL triggers or rewrite rules. Cumbersome. Inefficient.)
UPDATE/INSERTION/DELETION ANOMALIES

- Recall table instructions and embedded FD \{ \text{img} \} \rightarrow \{ \text{width}, \text{height} \}:

  \begin{array}{cccccccccccc}
  & \text{instructions} & \text{set} & \text{step} & \text{piece} & \text{color} & \text{quantity} & \text{page} & \text{img} & \text{width} & \text{height} \\
  \end{array}

- **Update anomaly:**
  Changing a single mini-world fact requires the modification of multiple rows.
  [ Modifying image size requires to search/update entire instruction table. ]

- **Insertion anomaly:**
  A new mini-world fact cannot be stored unless it is put in larger context.
  [ No place to record width/height dimension of a new image yet unused in an instruction manual. ]

- **Deletion anomaly:**
  A formerly stored mini-world fact vanishes once its (last) context is deleted.
  [ Information about image width/height is lost once last instruction manual including that image is deleted from instructions. ]
Boyce-Codd Normal Form (BCNF)

Table $R$ is in Boyce-Codd Normal Form (BCNF) if and only if all its FDs are already implied by its key constraints.

For table $R$ in BCNF and any FD $\beta \rightarrow c$ of $R$ one of the following holds:

1. The FD is trivial, i.e., $c \in \beta$.
2. The FD follows from a key because $\beta$ (or a subset of it) already is a key of $R$.

- A table in BCNF does not exhibit the three anomalies (no embedded functions).
- All FDs in table in BCNF are protected by the RDBMS through \texttt{PRIMARY KEY} (or \texttt{UNIQUE}) constraints.
BOYCE-CODD NORMAL FORM

- **Examples:**
  - Table *instructions* is not in BCNF: key FD \{ *set*, *step*, *piece*, *color* \} → \{ *quantity*, *page*, *img*, *width*, *height* \} does not imply \{ *set*, *step* \} → \{ *page*, *img* \} or \{ *img* \} → \{ *width*, *height* \}:

    | instructions | *set* | *step* | *piece* | *color* | *quantity* | *page* | *img* | *width* | *height* |
    |--------------|------|-------|--------|--------|-----------|------|-----|--------|---------|

  - Table *users* not in BCNF: \{ *rating* \} → \{ *stars* \} not implied by key FD:

    | users | *name* | *rating* | *stars* |
    |-------|--------|---------|--------|

  - Table *stores* not in BCNF: \{ *state* \} → \{ *country* \} not implied by key FD:

    | stores | *store* | *city* | *state* | *country* |
    |--------|--------|-------|--------|---------|
**BCNF SCHEMA DECOMPOSITION**


- If $\beta \rightarrow c \in F$ with $c \notin \beta$ and $\beta$ does not contain a key of $R$ then
  1. Split and replace $R$ by
     - $R_1((sch(R) \setminus cover(\beta, F)) \cup \beta)$
     - $R_2(cover(\beta, F))$
  2. split($R_1, F_{sch(R_1)}$)
  3. split($R_2, F_{sch(R_2)}$)

- Notes:
  - $F_C$ denotes FD set $F$ restricted to those $\beta \rightarrow c$ for which $\beta \cup \{c\} \subseteq C$.
  - For each split: $sch(R_1) \cup sch(R_2) = sch(R)$ and $sch(R_1) \cap sch(R_2) = \beta$. 

- Resultant BCNF tables after $\text{split}(\text{instructions}, F')$ has been completed:

<table>
<thead>
<tr>
<th>parts (1/3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>set</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>:</td>
</tr>
<tr>
<td>9495–1</td>
</tr>
<tr>
<td>9495–1</td>
</tr>
<tr>
<td>9495–1</td>
</tr>
<tr>
<td>9495–1</td>
</tr>
<tr>
<td>9495–1</td>
</tr>
<tr>
<td>9495–1</td>
</tr>
<tr>
<td>9495–1</td>
</tr>
</tbody>
</table>

- Note: It is rather straightforward to name the newly generated tables: these tables represent a single real-world concept.
BCNF: AFTER DECOMPOSITION (2)

- Resultant BCNF tables after $\text{split}(\text{instructions}, F')$ has been completed:

  \[
  \begin{array}{cccc}
  \text{layouts} (2/3) \\
  \text{set} & \text{step} & \text{page} & \text{img} \\
  \vdots & \vdots & \vdots & \vdots \\
  9495-1 & 7 & 24 & <\text{image07}> \\
  9495-1 & 8 & 24 & <\text{image08}> \\
  9495-1 & 9 & 24 & <\text{image09}> \\
  \vdots & \vdots & \vdots & \vdots \\
  \end{array}
  \]

  \[
  \begin{array}{cccc}
  \text{illustrations} (3/3) \\
  \text{img} & \text{width} & \text{height} \\
  \vdots & \vdots & \vdots \\
  <\text{image07}> & 639 & 533 \\
  <\text{image08}> & 650 & 522 \\
  <\text{image09}> & 541 & 638 \\
  \vdots & \vdots & \vdots \\
  \end{array}
  \]

- To tie the BCNF tables together, establish foreign keys pointing from parts to layouts and from layouts to illustrations.
BCNF: RECONSTRUCTION

- Use an **equi-join to reconstruct** the original wide table **instructions** from its constituent tables:

**Reconstruction after BCNF decomposition**

Perform an equi-join over the (non-empty) schema intersections of the BCNF tables:

```
SELECT p.set, p.step, p.piece, p.color, p.quantity,
     l.page, l.img,
     i.width, i.height
FROM parts p, layouts l, illustrations i
WHERE p.set = l.set AND p.step = l.step
    AND l.img = i.img
```

- It may make sense to use **CREATE VIEW** to reestablish the wide table for users and applications.
BCNF: AFTER DECOMPOSITION

- Decomposition for table **users**:

<table>
<thead>
<tr>
<th>user</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>3</td>
</tr>
<tr>
<td>Bert</td>
<td>1</td>
</tr>
<tr>
<td>Cora</td>
<td>4</td>
</tr>
<tr>
<td>Drew</td>
<td>5</td>
</tr>
<tr>
<td>Erik</td>
<td>1</td>
</tr>
<tr>
<td>Fred</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>user</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alex</td>
<td>3</td>
</tr>
<tr>
<td>Bert</td>
<td>1</td>
</tr>
<tr>
<td>Cora</td>
<td>4</td>
</tr>
<tr>
<td>Drew</td>
<td>5</td>
</tr>
<tr>
<td>Erik</td>
<td>1</td>
</tr>
<tr>
<td>Fred</td>
<td>3</td>
</tr>
</tbody>
</table>

- The RDBMS protects the FDs (keys): translation from rating to stars in table **render** is always consistent. No redundancy in table **render**.
BNCF decomposition builds on the assumption that no information is lost during the splits: original table $R$ can be reconstructed by an equi-join of $R_1$ and $R_2$.

Not all decompositions are lossless, however. Consider:

$$
\begin{array}{ccc}
A & B & C \\
\hline
a_1 & b_1 & c_1 \\
a_1 & b_1 & c_2 \\
a_1 & b_2 & c_1 \\
\end{array}
$$

and its decomposition into $R_1(A, B)$, $R_2(A, C)$. The equi-join of $R_1$ and $R_2$ (on $A$) is:

$$
\begin{array}{ccc}
A & B & C \\
\hline
a_1 & b_1 & c_1 \\
a_1 & b_1 & c_2 \\
a_1 & b_2 & c_1 \\
a_1 & b_2 & c_2 \\
\end{array}
$$

⇒ An extra (bogus!) row has been reconstructed by the join. Information has been lost.
BCNF: LOSSLESS SPLITS

Decomposition Theorem

Consider the decomposition of table $R$ into $R_1$ and $R_2$. The reconstruction of $R$ from $R_1$, $R_2$ via an equi-join on $sch(R_1) \cap sch(R_2)$ is lossless if

1. $sch(R_1) \cup sch(R_2) = sch(R)$ and
2. $sch(R_1) \cap sch(R_2)$ is a key of $R_1$ or $R_2$ (or both).

- The splits “along the FD $\beta \rightarrow c$” performed by $\text{split}(R,F)$ will always be lossless:
  1. $sch(R_1) \cup sch(R_2) = sch(R)$ and $sch(R_1) \cap sch(R_2) = \beta$
  2. Since $sch(R_2) = \text{cover}(\beta, F)$, $\beta$ is a key for $R_2$
- We will never lose information through BCNF decomposition.
BCNF: NON-DETERMINISM, LOSS OF FDS

-⚠️ BCNF is not deterministic: arbitrary choice of the “split FD” in algorithm $\text{split}(R, F)$ leads to different decompositions, in general:

1. For table instructions: splitting along FD $\{ \text{set, step} \} \rightarrow \{ \text{page, img} \}$ or $\{ \text{img} \} \rightarrow \{ \text{width, height} \}$ first makes no difference. (Try it.)

2. But consider $R(\text{A, B, C, D, E})$ with FDs $\{ \text{C, D} \} \rightarrow \text{E}$ and $\{ \text{B} \} \rightarrow \text{E}$.

-⚠️ BCNF decomposition may fail to preserve dependencies: given FD $\beta \rightarrow c$, the column set $\beta \cup \{ c \}$ may be distributed across multiple tables. The FD is “lost” (cannot be enforced by the system).

- Consider FDs $\{ \text{zip} \} \rightarrow \{ \text{city, state} \}$ and $\{ \text{street, city, state} \} \rightarrow \text{zip}$ in table zipcodes.

1. What are the candidate keys for zipcodes?
2. What is a BCNF decomposition for zipcodes?

<table>
<thead>
<tr>
<th>zipcodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>zip</td>
</tr>
</tbody>
</table>
DENORMALIZATION VS. DECOMPOSITION

- BCNF and decomposition come with significant benefits but are no panacea. There are valid reasons to leave database tables in denormalized form:

1. **Performance:**
   Decomposition requires table reconstruction via equi-joins which incur query evaluation costs. Denormalized table save this effort at the cost of storing information redundantly.

2. **Preservation of FDs:**
   In specific applications, preservation of mission-critical FDs may be a higher priority than the removal of redundancy.

- **Columnar database systems** perform **full decomposition** (beyond the splits required by BCNF normalization): \( R(\text{id}, A, B, C, \ldots) \) decomposed into \( R1(\text{id}, A), R2(\text{id}, B), R3(\text{id}, C), \ldots \) (binary tables).

- Queries other than `SELECT r.* FROM R r` can selectively access the \( R_i \), reading less bytes from persistent storage.

- DBMS internals simplified: every row is guaranteed to have exactly two fields.