

INTRODUCTION TO RELATIONAL DATABASE SYSTEMS

DATENBANKSYSTEME 1 (INF 3131)

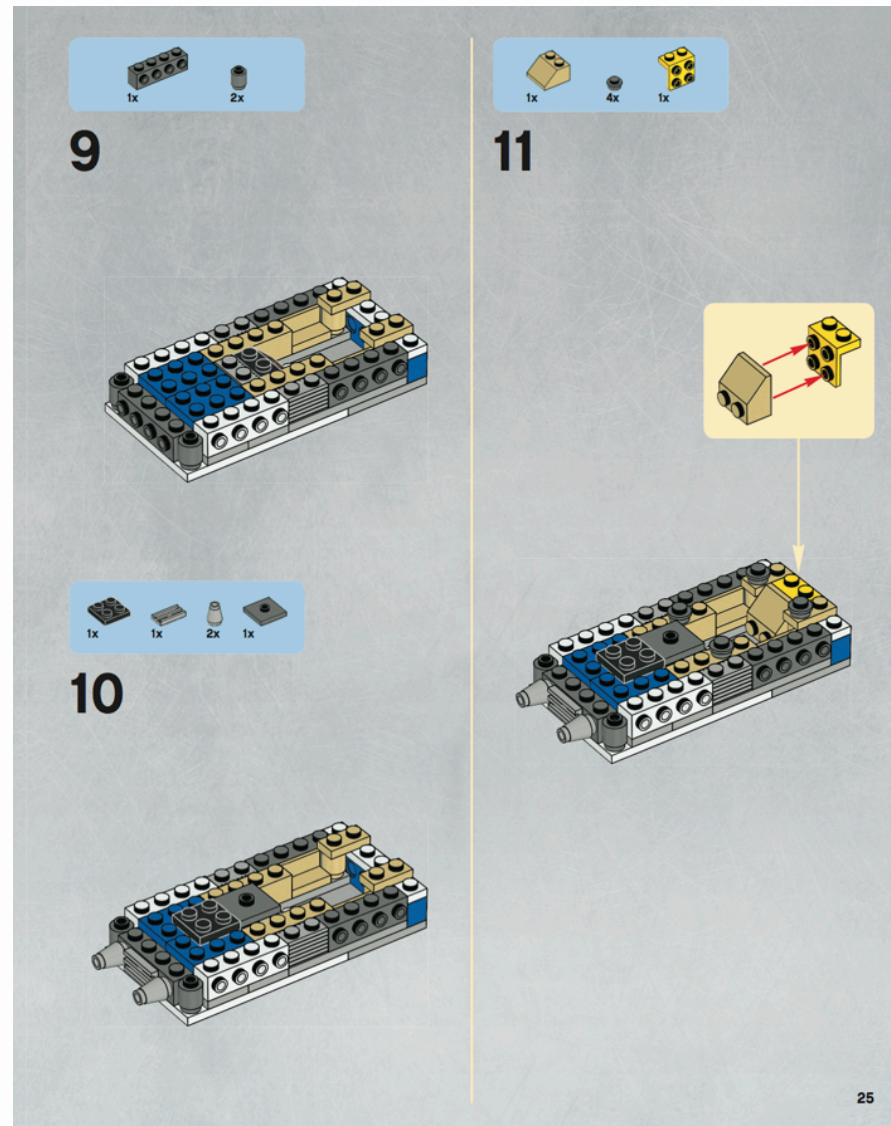
**Torsten Grust
Universität Tübingen**

Summer 2023

LEGO 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 *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



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 and y , then

$$x = y \wedge f(x) = z \Rightarrow f(y) = z$$

- 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, α) 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 β , they also agree on column c .” (β : function arguments, c : function result).

Notation: the FD $\beta \rightarrow \{c_1, \dots, c_n\}$ abbreviates the set of FDs $\beta \rightarrow c_1, \dots, \beta \rightarrow c_n$.

- 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

set	step	piece	color	quantity	page	img	width	height
:	:	:	:	:	:	:	:	:
9495-1	7	3010	2	2	24	<image07>	639	533
9495-1	7	3023	2	2	24	<image07>	639	533
9495-1	7	2877	86	1	24	<image07>	639	533
9495-1	8	3002	7	2	24	<image08>	650	522
9495-1	8	30414	1	2	24	<image08>	650	522
9495-1	9	30414	85	1	25	<image09>	541	638
9495-1	9	3062b	85	2	25	<image09>	541	638
:	:	:	:	:	:	:	:	:

- Which functional dependencies hold in table `instructions`?

FUNCTIONAL DEPENDENCIES

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

```
SELECT DISTINCT 'The FD  $\{b_1, \dots, b_n\} \rightarrow c$  does not hold'  
FROM   R AS r  
GROUP BY r.b1, ..., r.bn  
HAVING COUNT(DISTINCT r.c) > 1
```

Aggregate Functions

Optional modifier DISTINCT affects the computation of aggregate functions:

```
<aggregate>([ ALL ] expression)  -- aggregate all non-NULL values  
<aggregate>(DISTINCT expression) -- aggregate all distinct non-NULL values  
<aggregate>(*)                    -- aggregate all rows (count(*))
```

KEY \rightarrow FD

- 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 $(R, \{a_1, \dots, a_k, a_{k+1}, \dots, a_n\})$.

$$\{a_1, \dots, a_k\} \text{ is a key of } R \Leftrightarrow \\ \{a_1, \dots, a_k\} \rightarrow \{a_{k+1}, \dots, a_n\} \text{ holds.}$$

- So, keys are special FDs.
- Turning this around: FDs are a generalization of keys.

FD \rightarrow (LOCAL, PARTIAL) KEY

Keys vs FDs (2)

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

```
SELECT DISTINCT  $\beta$ ,  $c$ 
FROM  $R$ 
```

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

<u>set</u>	<u>step</u>	page
:	:	:
9495-1	7	24
9495-1	8	24
9495-1	9	25
9495-1	10	25
9495-1	11	25
:	:	:

(i.e., exactly the table materializing the function `printed_on()`, see above).

FUNCTIONAL DEPENDENCIES

- **Example:** recall table `stores` of the LEGO Data Warehouse scenario:

<u>store</u>	city	state	country
⋮	⋮	⋮	⋮
7	HAMBURG	Hamburg	Germany
8	LEIPZIG	Sachsen	Germany
9	MÜNCHEN	Bayern	Germany
10	MÜNCHEN PASING	Bayern	Germany
11	NÜRNBERG	Bayern	Germany
⋮	⋮	⋮	⋮
16	ARDEN FAIR MALL	CA	USA
17	DISNEYLAND RESORT	CA	USA
18	FASHION VALLEY	CA	USA
⋮	⋮	⋮	⋮

- 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:

users

<u>user</u>	rating	stars
Alex	3	***
Bert	1	*
Cora	4	****
Drew	5	*****
Erik	1	*
Fred	3	***

- FD {rating} → stars materializes the **computable function** $\text{stars} = f(\text{rating}) = \text{repeat}('*', \text{rating})$ [see PostgreSQL's string function library].
- In such cases, good database design should consider to **trade materialization for computation**. Removes redundancy.

SQL: VIEWS

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.

SQL: VIEWS

- 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 $\text{stars} = f(\text{rating})$ into a computed function:

```
-- 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) AS stars
  FROM   users AS u;
```

- Since PostgreSQL's `repeat()` is a pure function, the FD $\text{rating} \rightarrow \text{stars}$ 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 $c \in \text{sch}(R)$):

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 $\alpha \subseteq sch(R)$ of columns and a set of FDs F over R , compute the **cover** α^+ , i.e., the set of all columns functionally determined by α .

Cover

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

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

-  Should we find that $\alpha^+ = sch(R)$, then α is a super key for R .

DERIVING FDS (COVER)

- Compute the cover α^+ for a given set of FDs:

cover(α, F) (input: column set α , FD set F , output: α^+):

1 $X \leftarrow \alpha$

2 repeat

 for each FD $\beta \rightarrow c \in F$ do

 if $\beta \subseteq X$ then

$X \leftarrow X \cup \{c\}$

until X did not change

3 return X

DERIVING FDS (COVER)

- **Example:** In table `instructions`, compute $\{\text{set}, \text{step}\}^+$ with $F = \{ \{\text{set}, \text{step}\} \rightarrow \text{page}, \{\text{set}, \text{step}\} \rightarrow \text{img}, \{\text{img}\} \rightarrow \text{width}, \{\text{img}\} \rightarrow \text{height} \}$.

`instructions`

<code>set</code>	<code>step</code>	<code>piece</code>	<code>color</code>	<code>quantity</code>	<code>page</code>	<code>img</code>	<code>width</code>	<code>height</code>
------------------	-------------------	--------------------	--------------------	-----------------------	-------------------	------------------	--------------------	---------------------

- Tracing column set X :

① $X := \{\text{set}, \text{step}\}$

② FD $\{\text{set}, \text{step}\} \rightarrow \text{page}, \{\text{set}, \text{step}\} \subseteq X$: $X := X \cup \{\text{page}\}$
FD $\{\text{set}, \text{step}\} \rightarrow \text{img}, \{\text{set}, \text{step}\} \subseteq X$: $X := X \cup \{\text{img}\}$
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. $X = \{\text{set}, \text{step}, \text{page}, \text{img}, \text{width}, \text{height}\}$
Repetition of ② does not add new columns to X .

③ Return $\{\text{set}, \text{step}, \text{page}, \text{img}, \text{width}, \text{height}\}$.

DERIVING CANDIDATE KEYS

key(K, U, F) (input: FD set F , output: set of all candidate keys for R)

```
if  $U = \emptyset$  then                                     } invariant: cover( $K \cup U, F$ ) =  $sch(R)$ 
| return  $\{K\}$ 
else
|  $X \leftarrow \emptyset$ 
| for each  $c \in U$  do
|   if  $c \notin \mathbf{cover}(K \cup (U \setminus \{c\}), F)$  then } is  $c$  essential for the key?
|   |  $X \leftarrow X \cup \mathbf{key}(K \cup \{c\}, U \setminus \{c\}, F)$   ┆
|   | else
|   |    $X \leftarrow X \cup \mathbf{key}(K, U \setminus \{c\}, F)$ 
| return  $X$ 
```

- Invoke via **key**($\emptyset, sch(R), F$).
- Can optimize at ┆: invoke **key**($K \cup \{c\}, U \setminus \mathbf{cover}(K \cup \{c\}, F), F$) instead.

DATABASE DESIGN WITH FDS

- 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 `{img} → {width, height}`:

`instructions`

<code>set</code>	<code>step</code>	<code>piece</code>	<code>color</code>	<code>quantity</code>	<code>page</code>	<code>img</code>	<code>width</code>	<code>height</code>
------------------	-------------------	--------------------	--------------------	-----------------------	-------------------	------------------	--------------------	---------------------

- **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

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 β (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 **PRIMARY KEY** (or **UNIQUE**) constraints.

BOYCE-CODD NORMAL FORM

- Examples:

- Table `instructions` is not in BCNF: key FD $\{\text{set}, \text{step}, \text{piece}, \text{color}\} \rightarrow \{\text{quantity}, \text{page}, \text{img}, \text{width}, \text{height}\}$ does not imply $\{\text{set}, \text{step}\} \rightarrow \{\text{page}, \text{img}\}$ or $\{\text{img}\} \rightarrow \{\text{width}, \text{height}\}$:

`instructions`

<u>set</u>	<u>step</u>	<u>piece</u>	<u>color</u>	quantity	page	img	width	height
------------	-------------	--------------	--------------	----------	------	-----	-------	--------

- Table `users` not in BCNF: $\{\text{rating}\} \rightarrow \{\text{stars}\}$ not implied by key FD:

`users`

<u>name</u>	rating	stars
-------------	--------	-------

- Table `stores` not in BCNF: $\{\text{state}\} \rightarrow \{\text{country}\}$ not implied by key FD:

`stores`

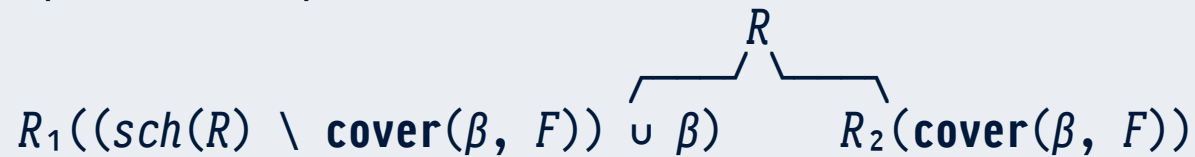
<u>store</u>	city	state	country
--------------	------	-------	---------

BCNF SCHEMA DECOMPOSITION

split(R, F) (input: table R with FD set F output: splitted relation schemata)

if $\beta \rightarrow c \in F$ with $c \notin \beta$ and β does not contain a key of R then

split and replace R :



split($R_1, F|sch(R_1)$)

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 $sch(R_1) \cap sch(R_2) = \beta$.

BCNF: AFTER DECOMPOSITION (1)

- Resultant BCNF tables after `split(instructions, F)` has been completed:

parts (1/3)

<u>set</u>	<u>step</u>	<u>piece</u>	<u>color</u>	<u>quantity</u>
⋮	⋮	⋮	⋮	⋮
9495-1	7	3010	2	2
9495-1	7	3023	2	2
9495-1	7	2877	86	1
9495-1	8	3002	7	2
9495-1	8	30414	1	2
9495-1	9	30414	85	1
9495-1	9	3062b	85	2
⋮	⋮	⋮	⋮	⋮

- 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 `split(instructions, F)` has been completed:

layouts (2/3)

<u>set</u>	<u>step</u>	<u>page</u>	<u>img</u>
⋮	⋮	⋮	⋮
9495-1	7	24	<image07>
9495-1	8	24	<image08>
9495-1	9	24	<image09>
⋮	⋮	⋮	⋮

illustrations (3/3)

<u>img</u>	<u>width</u>	<u>height</u>
⋮	⋮	⋮
<image07>	639	533
<image08>	650	522
<image09>	541	638
⋮	⋮	⋮

- 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 AS p, layouts AS l, illustrations AS 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`:

`users` (R_1)

<u>user</u>	rating
Alex	3
Bert	1
Cora	4
Drew	5
Erik	1
Fred	3

`render` (R_2)

<u>rating</u>	stars
1	*
3	***
4	****
5	*****

- The RDBMS protects the FDs (keys): translation from rating to stars in table `render` is always consistent. No redundancy in table `render`.

BCNF DECOMPOSITION: LOSSLESS SPLITS

- BCNF 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:

R

<u>A</u>	<u>B</u>	<u>C</u>
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_2	c_1

and its decomposition into $R_1(A, B)$, $R_2(A, C)$. This is a somewhat arbitrary split, *not* suggested by an FD. The equi-join of R_1 and R_2 (on A) is:

<u>A</u>	<u>B</u>	<u>C</u>
a_1	b_1	c_1
a_1	b_1	c_2
a_1	b_2	c_1
a_1	b_2	c_2

⇒ 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 $\text{sch}(R_1) \cap \text{sch}(R_2)$ is **lossless** if

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

- The splits “along the FD $\beta \rightarrow c$ ” performed by **split**(R, F) will always be lossless:
 1. $\text{sch}(R_1) \cup \text{sch}(R_2) = \text{sch}(R)$ and $\text{sch}(R_1) \cap \text{sch}(R_2) = \beta$ ✓
 2. Since $\text{sch}(R_2) = \text{cover}(\beta, F)$, β 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 `split(R,F)` leads to different decompositions, in general:
 1. For table `instructions`: splitting along FD `{set,step} → {page, img}` or `{img} → {width,height}` first makes no difference. (Try it.)
 2. But consider `R(A,B,C,D,E)` with FDs `{C,D} → E` and `{B} → 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 `{zip} → {city, state}` and `{street, city, state} → zip` in table `zipcodes`.
 1. What are the candidate keys for `zipcodes`?
 2. What is a BCNF decomposition for `zipcodes`?

`zipcodes`

<code>zip</code>	<code>street</code>	<code>city</code>	<code>state</code>
------------------	---------------------	-------------------	--------------------

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): $R(\underline{id}, A, B, C, \dots)$ decomposed into $R_1(\underline{id}, A)$, $R_2(\underline{id}, B)$, $R_3(\underline{id}, C)$, ... (binary tables).
 - Queries other than `SELECT r.* FROM R AS 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.