Technische Hochschule Rosenheim

Chapter 4 – Relational database design

Databases lectures

Prof. Dr Kai Höfig

Handout

RWINES

Name	Colour	Vintage	Vineyard	Growing_area	Region
La Rose GrandCru	Red	1998	Chateau La Rose	Saint-Emilion	Bordeaux
Creek Shiraz	Red	2003	Creek	Barossa Valley	South Australia
Zinfandel	Red	2004	Helena	Napa Valley	California
Pinot Noir	Red	2001	Creek	Barossa Valley	South Australia
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	Rheingau	Hesse
Chardonnay	White	2002	Bighorn	Napa Valley	California

PRODUCER

Vineyard	Growing_area	Region
Creek	Barossa Valley	South Australia
Helena	Napa Valley	California
Chateau La Rose	Saint-Emilion	Bordeaux
Chateau La Pointe	Pomerol	Bordeaux
Müller	Rheingau	Hesse
Bighorn	Napa Valley	California

WINES

Name	Colour	Vintage	Vineyard
La Rose GrandCru	Red	1998	Chateau La Rose
Creek Shiraz	Red	2003	Creek
Zinfandel	Red	2004	Helena
Pinot Noir	Red	2001	Creek
Pinot Noir	Red	1999	Helena
Riesling Reserve	White	1999	Müller
Chardonnay	White	2002	Bighorn

Where does the schema of a DB actually come from?

- Example: wine management for a wine merchant.

RWINES

Name	**Colour**	**Vintage**	**Vineyard**	**Growing_area**	**Region**
La Rose GrandCru	Red	1998	Chateau La Rose	Saint-Emilion	Bordeaux
Creek Shiraz	Red	2003	Creek	Barossa Valley	South Australia
Zinfandel	Red	2004	Helena	Napa Valley	California
Pinot Noir	Red	2001	Creek	Barossa Valley	South Australia
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	Rheingau	Hesse
Chardonnay	White	2002	Bighorn	Napa Valley	California

- What problems are there with change operations on this table?

Overview

- 4.1 Redundancy & anomalies
- 4.2 Functional dependencies and keys
- 4.3 Closure, membership, CLOSURE and COVER
- 4.4 Normal forms
- 4.5 Transformations and decomposition

Redundancies

Redundancies in relations are undesirable for several reasons

- Main problem: redundancies lead to anomalies
  - Update anomalies: **change operations** on relations with redundancies are difficult to implement correctly: if redundant information occurs, an update must change that information in all its occurrences
    → difficult to implement with normal relational change operations and the local integrity constraints (keys) that occur in relational systems
    → occur with INSERT or UPDATE
  - Deletion anomalies: **delete operations** can lead to the unintended loss of additional information
    → occur with DELETE
- Less critical problem: redundant information takes up unnecessary **storage space**

Example of an update anomaly

- Update anomaly
  - Insertion of inconsistent information into the `RWINES` relation:

```
        insert into RWINES(Name, Colour, Vintage,
                           Vineyard, Growing_area, Region)
          values           ('Chardonnay', 'White', 2004,
                           'Helena', 'Rheingau', 'California')
```

  - The `Helena` vineyard was previously located in `Napa Valley`, but now the database contains both pieces of information (`Napa Valley` and `Rheingau`)
  - `Rheingau` was previously located in `Hesse`, but now it is also located in `California`

RWINES

Name	**Colour**	**Vintage**	**Vineyard**	**Growing_area**	**Region**
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	Rheingau	Hesse
Chardonnay	White	2002	Helena	Rheingau	California

Example of a deletion anomaly

- Deletion anomaly
  - Delete the information that the `'Chateau La Rose'` vineyard produces a wine called `'La Rose GrandCru'`

```
        delete from  RWINES
           where Name = 'La Rose GrandCru'
```

  - Loses (unintentionally) the information that `Saint-Emilion` is in `Bordeaux`.

RWINES

Name	Colour	Vintage	Vineyard	Growing_area	Region
~~La Rose GrandCru~~	~~Red~~	~~1998~~	~~Chateau La Rose~~	~~Saint-Emilion~~	~~Bordeaux~~
Creek Shiraz	Red	2003	Creek	Barossa Valley	South Australia
Zinfandel	Red	2004	Helena	Napa Valley	California
Pinot Noir	Red	2001	Creek	Barossa Valley	South Australia
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	Rheingau	Hesse
Chardonnay	White	2002	Bighorn	Napa Valley	California

Avoiding anomalies

Avoiding anomalies in databases

Step 1: Identifying redundancies

Step 2: Eliminating redundancies

Step 1: Identifying redundancies (1)

Redundancies arise from the application – from the knowledge about the data

- Examples of "knowledge about the data"
  - Every `Vineyard` is located in exactly one `Growing_area`, i.e. there must not be two tuples with the same `Vineyard` but a different `Growing_area`.
  - Every `Growing_area` is located in exactly one `Region`, i.e. if two tuples have the same `Growing_area`, then the `Region` must also be the same.
- General info:

  If the values for attribute *A* are the same for two tuples, then the values for attribute *B* must also be the same.

RWINES

Name	**Colour**	**Vintage**	**Vineyard**	**Growing_area**	**Region**
La Rose GrandCru	Red	1998	Chateau La Rose	Saint-Emilion	Bordeaux
Creek Shiraz	Red	2003	Creek	Barossa Valley	South Australia
Zinfandel	Red	20	[illegible]	[illegible]	nia
Pinot Noir	Red	2001	Creek	Barossa Valley	South Australia
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	[illegible]	[illegible]
Chardonnay	White	2002	Bighorn	Napa Valley	California

Vineyard determines $\rightarrow$ Growing_area

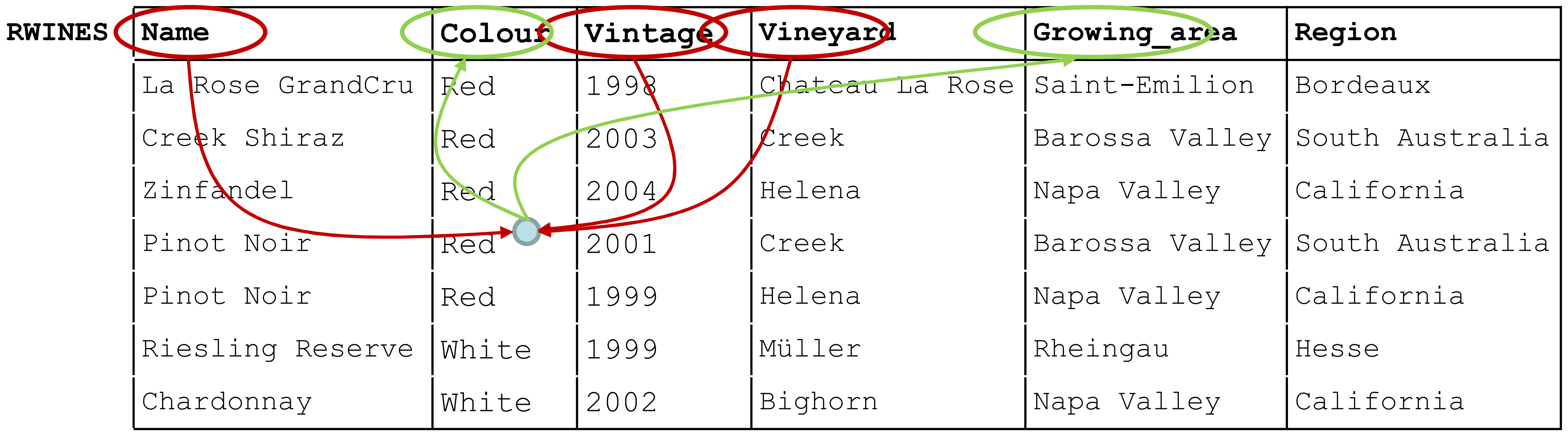
Growing_area determines $\rightarrow$ Region

Step 1: Identifying redundancies (2)

Redundancies arise from the application – from the knowledge about the data

- A more complex example: if the `Name`, `Vintage` and `Vineyard` are the same, then the `Growing_area` and `Colour` must also be the same.
- General info:

  If the values for attributes $A_1, A_2, \ldots, A_n$ are the same,
  then the values for attributes $B_1, B_2, \ldots, B_m$ must also be the same



RWINES

Name	**Colour**	**Vintage**	**Vineyard**	**Growing_area**	**Region**
La Rose GrandCru	Red	1998	Chateau La Rose	Saint-Emilion	Bordeaux
Creek Shiraz	Red	2003	Creek	Barossa Valley	South Australia
Zinfandel	Red	2004	Helena	Napa Valley	California
Pinot Noir	Red	2001	Creek	Barossa Valley	South Australia
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	Rheingau	Hesse
Chardonnay	White	2002	Bighorn	Napa Valley	California

Step 1: Identifying redundancies (3)

Functional dependencies

- If the values for attributes $A = A_1, A_2, \ldots, A_n$ are the same, then the values for attributes $B = B_1, B_2, \ldots, B_m$ must also be the same
- We write $A \rightarrow B$ and mean that A *uniquely determines* B or B *is functionally dependent on A.*
- Functional dependencies (FDs) can only be determined from the context of the respective database application.
- The **user** knows the FDs and the task of the **database designer** is to determine all or at least a representative set (more on this later) of all FDs.
- This process cannot be automated, because an FD $A \rightarrow B$ must not only apply to the current set of tuples, but also to all possible future tuples.

  - Example of wine database: in a conversation between a wine connoisseur (user, subject expert) and the database designer, the wine connoisseur mentions that there are no growing areas that are spread across multiple regions. The attentive database designer immediately notes the FD Growing_area → Region.

Step 2: Eliminating redundancies (1)

- Example of a functional dependency from before:
  Every `Vineyard` is located in exactly one `Growing_area`, i.e. there must not be two tuples with the same `Vineyard` but a different `Growing_area`.

RWINES

Name	**Colour**	**Vintage**	**Vineyard**	**Growing_area**	**Region**
La Rose GrandCru	Red	1998	Chateau La Rose	Saint-Emilion	Bordeaux
Creek Shiraz	Red	2003	Creek	Barossa Valley	South Australia
Zinfandel	Red	2004	Helena	Napa Valley	California
Pinot Noir	Red	2001	Creek	Barossa Valley	South Australia
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	Rheingau	Hesse
Chardonnay	White	2002	Bighorn	Napa Valley	California

PRODUCER

Vineyard	**Growing_area**	**Region**
Creek	Barossa Valley	South Australia
Helena	Napa Valley	California
Chateau La Rose	Saint-Emilion	Bordeaux
Chateau La Pointe	Pomerol	Bordeaux
Müller	Rheingau	Hesse
Bighorn	Napa Valley	California

WINES

Name	**Colour**	**Vintage**	**Vineyard**
La Rose GrandCru	Red	1998	Chateau La Rose
Creek Shiraz	Red	2003	Creek
Zinfandel	Red	2004	Helena
Pinot Noir	Red	2001	Creek
Pinot Noir	Red	1999	Helena
Riesling Reserve	White	1999	Müller
Chardonnay	White	2002	Bighorn

➔ This is called **normalisation**

Step 2: Eliminating redundancies (2)

- Example of a functional dependency from before:
  Every `Vineyard` is located in exactly one `Growing_area`, i.e. there must not be two tuples with the same `Vineyard` but a different `Growing_area`.

- How do we ensure such a thing in a relational database?

- First idea: `Vineyard` must be a key of the table!
  - Then the DBMS ensures that there are no two tuples with the same `Vineyard`

- But: `Vineyard` cannot be a key of `RWINES`, because then we can only store one wine in `RWINES` for each `Vineyard`.

- Solution: splitting `RWINES` into two relations – `WINES` and `PRODUCER`.

- So far so good, but we have even more functional dependencies (e.g. there is exactly one `Region` for each `Growing_area`), so we have to do this for all functional dependencies (i.e.: splitting relations further)

➔ This is called **normalisation**

Summary

- There are many ways to create a schema for a relational DB
  - e.g. from an ER diagram or directly from the requirements analysis
- However, the first schema often needs improvement because it contains redundancies.
- Redundancies lead to anomalies (and wasted storage space).
- Functional dependencies describe application knowledge about the data. And they allow us to identify and eliminate redundancies.
- Normal forms theory: constraints formulated by functional dependencies that a schema must meet so that it does not contain redundancies and thus does not cause any anomalies.
- Normalisation: decomposition (splitting) of the relations into two or more new relations with the help of functional dependencies, which no longer contain corresponding redundancies and thus do not cause any anomalies.

Revision: Definitions in the relational model

- $R = \{A_1,..,A_k\}$ a **relational schema** about the attributes A_i
  - `PRODUCER` = { `Vineyard, Growing_area, Region`}
- $S = \{R_1,.., R_m\}$ a **database schema** about the relational schema R_i
  - *WineDB* = {`WINES, PRODUCER`}
- *r(R)* a **relation** about the relational schema R consisting of **tuples** $t_1(A_1),..,t_i(A_k)$ about attributes A_j
  - r(`PRODUCER`) = $\{t_1, t_2, t_3\}$ with
    t_1(`Vineyard`)=`'Creek'`
    t_1(`Growing_area`)=`'Barossa Valley'`
    t_1(`Region`)=`'South Australia' and so on`
- $d:=\{r_1,..,r_P\}$ a **database** about the relations r_i

Identifier attribute set

- An **identifier attribute set** is a set of attributes $\{B_1, \dots, B_k\} \subseteq R$ so that two different tuples about R always differ in at least one attribute value $B \in \{B_1, \dots, B_k\}$. Formally:
  - $\forall t_1, t_2 \in r \,.\, [\; t_1 \neq t_2 \;\Rightarrow\; \exists B \in \{B_1, \dots, B_k\} : t_1(B) \neq t_2(B) \;]$

  Or put another way: if we look at the tuples only using B, there are no identical rows.

  Example: {Name, Vintage, Vineyard}, **{Name, Colour, Vintage, Vineyard}**

RWINES

Name	**Colour**	**Vintage**	**Vineyard**	**Growing_area**	**Region**
La Rose GrandCru	Red	1998	Chateau La Rose	Saint-Emilion	Bordeaux
Creek Shiraz	Red	2003	Creek	Barossa Valley	South Australia
Zinfandel	Red	2004	Helena	Napa Valley	California
Pinot Noir	Red	2001	Creek	Barossa Valley	South Australia
Pinot Noir	Red	1999	Helena	Napa Valley	California
Riesling Reserve	White	1999	Müller	Rheingau	Hesse
Chardonnay	White	2002	Bighorn	Napa Valley	California

Key

- A **minimal** identifier attribute set is called a **key**.
  - Prime attribute: attribute of a key
  - Primary key: key specified when designing
  - Identifier attribute sets are also called superkeys (because they are supersets of a key)

  - Example: {`Name, Vintage, Vineyard`} is (the only) key for `WINES`

RWINES

Name	**Vintage**	**Vineyard**	**Growing_area**	**Region**
La Rose GrandCru	1998	Chateau La Rose	Saint-Emilion	Bordeaux
Creek Shiraz	2003	Creek	Barossa Valley	South Australia
Zinfandel	2004	Helena	Napa Valley	California
Pinot Noir	2001	Creek	Barossa Valley	South Australia
Pinot Noir	1999	Helena	Napa Valley	California
Riesling Reserve	1999	Müller	Rheingau	Hesse
Chardonnay	2002	Bighorn	Napa Valley	California

Foreign key

- If every value that occurs for an attribute *X* in a relation R_1 also occurs for an attribute *Y* in a relation R_2, then X is called a **foreign key** in R_1 for *Y* in R_2. Formally:

- When $X \subseteq R_1$, $Y \subseteq R_2$. The foreign key constraint $X(R_1) \rightarrow Y(R_2)$ is met if the following applies:

$$\{t(X) \mid t \in r_1\} \subseteq \{t(Y) \mid t \in r_2\}$$

- Example:
  $X = Y = \{$`Vineyard`$\}$ is a foreign key in `WINES` with respect to `PRODUCER`
  because the foreign key constraint is met:

  `Vineyard(WINES)` $\rightarrow$ `Vineyard(PRODUCER)`, i.e.
  $\{t(\{\texttt{Vineyard}\}) \mid t \in r_1(\texttt{WINES})\} \subseteq \{t(\{\texttt{Vineyard}\}) \mid t \in r_2(\texttt{PRODUCER})\}$

  in words: every value that occurs for `Vineyard` in `WINES` must also occur as a value for `Vineyard` in `PRODUCER`.

Functional dependencies - formally

- Functional dependency (FD) between attribute sets $A_1, A_2, \dots, A_n$ and $B_1, B_2, \dots, B_m$ of a relational schema R: if in every relation about R the values of the $A_1, A_2, \dots, A_n$ attributes determine the values of the $B_1, B_2, \dots, B_m$ attributes. $A \rightarrow A$ also applies.

- If two tuples do not differ in the $A_1, A_2, \dots, A_n$ attributes, then they also have the same values for all $B_1, B_2, \dots, B_m$ attributes

- Notation for functional dependency: $\{A_1, A_2, \dots, A_n\} \rightarrow \{B_1, B_2, \dots, B_m\}$ (set brackets for attribute sets are often omitted)
  - Example for `RWINES`:
    1) `{Name, Vintage, Vineyard}` → `{Growing_area, Colour}`
    2) `Growing_area` → `Region`
    3) but not: `Vineyard` → `Name`

- **Note: FDs say something about *all possible* database contents, not just the current content**

Relationship between functional dependencies and keys

- A set of attributes $K \subseteq R$ is a **key** of R, if and only if
  1. $K \rightarrow R$
  2. There is no real subset of $X \subset K$ with $X \rightarrow R$

  In words: the attributes in K uniquely determine all attributes of R and are minimal.

- A set of attributes $S \subseteq R$ is a **superkey** of R if and only if $S \rightarrow R.$

- This means: if we find functional dependencies where a whole relation is on the right, then we have found a (super)key.

Example of keys

- In the relation `PRODUCER` the following applies
  `Vineyard` $\rightarrow$ `Vineyard, Growing_area, Region`
  Trivially, there is no real subset of `{Vineyard}` which functionally determines `PRODUCER`. Thus `{Vineyard}` is a key for `PRODUCER`

  The superkeys are `{Vineyard,Region}`, `{Vineyard,Growing_area}` and `{Vineyard,Growing_area,Region}`

PRODUCER

<u>Vineyard</u>	**Growing_area**	**Region**
Creek	Barossa Valley	South Australia
Helena	Napa Valley	California
Chateau La Rose	Saint-Emilion	Bordeaux
Chateau La Pointe	Pomerol	Bordeaux
Müller	Rheingau	Hesse
Bighorn	Napa Valley	California

So how does that actually help us?

- If we know *all* the FDs that fulfil a relation, we can determine the keys of the relation!
  - Until now, keys were always "simply there", but now we have an idea where they actually come from!
  - Unfortunately, we usually don't know *all* FDs, but only a few. We therefore need rules to determine ("calculate") further (or all) FDs from given FDs.

- FDs and redundancies (and thus anomalies) are related
  - → We will formulate constraints using the FDs so that a relation which meets these constraints is free of any update anomalies and deletion anomalies
  - → We will also learn about an algorithm which allows us to use FDs to split a given relation that does not meet these constraints into multiple relations that meet the constraints
  - However, we also need rules for the transformation of ("calculation" with) FDs

Derivation of FDs

- Example of derivation of FDs:
  - Relation `R(A,B,C)` with the functional dependencies
    $A \rightarrow B$
    $B \rightarrow C$.
  - Possible version:

R	A	B	C
	a1	b1	c1
	a2	b1	c1
	a3	b2	c1
	a4	b1	c1

  - Obviously, the following also applies: $A \rightarrow C$ but not: $C \rightarrow A$ or $C \rightarrow B$

- General questions
  - Given a set of FDs F, which other FDs f can be derived from F?
  - Or even more interesting: given a set of FDs F, can a specific FD f be derived from F (membership problem)?

Transitivity

- Functional dependencies are transitive, which means that the following applies for $X, Y, Z \subseteq R$:

$$\{ X \rightarrow Y,\, Y \rightarrow Z \} \quad \Rightarrow \quad X \rightarrow Z$$

- Proof:
  - Assumption: in $r(R)$ the following applies:
    (1) $X \rightarrow Y$ and
    (2) $Y \rightarrow Z$
  - Therefore, for any two tuples $t_1, t_2 \in r(R)$ with $t_1(X) = t_2(X)$, the following must apply:
    (3) $t_1(Y) = t_2(Y)$ (because of (1))
    (4) $t_1(Z) = t_2(Z)$ (because of (3) and (2))
  - Therefore, the following applies: $X \rightarrow Z$

R

A	B	C
a1	b1	c1
a2	b1	c1
a3	b2	c1
a4	b1	c1

$$A \rightarrow B,\, B \rightarrow C \;\Rightarrow\; A \rightarrow C$$

Reflexivity

- Functional dependencies are reflexive, which means that for $Y \subseteq X \subseteq R$ the following applies:

$$X \supseteq Y \Rightarrow X \rightarrow Y$$

- Proof:
  - Assumption: $Y \subseteq X \subseteq R$;
    $t_1, t_2 \in r(R)$ with $t_1(X) = t_2(X)$
  - Then follows: $T_1(Y) = t_2(Y)$ because $X \supseteq Y$, and from this follows: $X \rightarrow Y$

R

A	**B**	**C**
a1	b1	c1
a2	b1	c1
a3	b2	c1
a4	b1	c1

$$A \subseteq AB \subseteq R \Rightarrow AB \rightarrow A$$

Augmentation

- The following applies to functional dependencies:

  $$\{X \rightarrow Y\} \quad \Rightarrow \; XZ \rightarrow YZ \text{ as well as } XZ \rightarrow Y$$

- Proof:
  - Assumption:
    $X \rightarrow Y$ applies in $r(R)$
    $XZ \rightarrow YZ$ does not apply.
  - There are two tuples t_1, $t_2 \in r(R)$ for which the following applies:
    $t_1(X) = t_2(X) \Rightarrow \; t_1(Y) = t_2(Y)$
    $t_1(XZ) = t_2(XZ) \Rightarrow t_1(Z) = t_2(Z)$
    $t_1(XZ) = t_2(XZ) \Rightarrow t_1(YZ) \neq t_2(YZ)$ contradiction.

R

A	B	C
a1	b1	c1
a1	b1	c2
a3	b2	c3
a4	b1	c4

R[A,C]

A	C
a1	c1
a1	c2
a3	c3
a4	c4

R[B,C]

B	C
b1	c1
b1	c2
b2	c3
b1	c4

$A \rightarrow B \;\Rightarrow\; AC \rightarrow BC$ and $AC \rightarrow B$

Derivation rules

- Requirement for a reasonable set of derivation rules
  - sound: rules do not derive FDs which are not logically implied
  - complete : all implied FDs are derived
  - independent (or minimal): no rule can be omitted

- Generally applicable rule set (sound & complete)
  - F1 Reflexivity $X \supseteq Y \quad \Rightarrow \quad X \to Y$
  - F2 Augmentation $\{ X \to Y \} \quad \Rightarrow \quad XZ \to YZ$ as well as $XZ \to Y$
  - F3 Transitivity $\{ X \to Y, Y \to Z \} \quad \Rightarrow \quad X \to Z$
  - F4 Decomposition $\{ X \to YZ \} \quad \Rightarrow \quad X \to Y$
  - F5 Union $\{ X \to Y, X \to Z \} \quad \Rightarrow \quad X \to YZ$
  - F6 Pseudotransitivity $\{ X \to Y, WY \to Z \} \quad \Rightarrow \quad WX \to Z$

- Comments
  - F1-F3 are known as Armstrong’s axioms (sound, complete, independent)
  - FDs such as in F1 are also known as trivial FDs.
  - F4 is also known as the splitting rule and F5 as the combining rule

Simplification of FD sets using the rules

- Given is a set F of FDs. We transform this in two steps:
  - 1st step: application of the splitting rule F4: Replace every FD

    $$A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m \in F$$

    with the m FDs

    $$\begin{aligned} &A_1, A_2, \ldots, A_n \rightarrow B_1 \\ &A_1, A_2, \ldots, A_n \rightarrow B_2 \\ &\ldots \\ &A_1, A_2, \ldots, A_n \rightarrow B_m \end{aligned}$$

  - 2nd step: elimination of trivial FDs according to F1

    Remove all FDs $A_1, A_2, \ldots, A_n \rightarrow B \in F$ with $B \in \{A_1, A_2, \ldots, A_n\}$ from F.

So from now on we can assume (implicitly) that every FD set F has this simplified form, and call this algorithm SPLITTING(F)

Example of simplification of FDs

Let's once again look at the relation

- `RWINES = {Name, Colour, Vintage, Vineyard, Growing_area, Region}`
- The following FDs are given
  1) `Name, Vintage, Vineyard` → `Growing_area, Colour`
  2) `Vineyard` → `Growing_area, Vineyard, Region`
  3) `Growing_area` → `Region`
- Simplification step 1 (splitting):
  1a) `Name, Vintage, Vineyard` → `Growing_area`
  1b) `Name, Vintage, Vineyard` → `Colour`
  2a) `Vineyard` → `Growing_area`
  2b) `Vineyard` → `Vineyard`
  2c) `Vineyard` → `Region`
  3) `Growing_area` → `Region`
- Simplification step 2 (elimination):
  Eliminate 2b)
- Results:
  1) `Name, Vintage, Vineyard` → `Growing_area`
  2) `Name, Vintage, Vineyard` → `Colour`
  3) `Vineyard` → `Growing_area`
  4) `Vineyard` → `Region`
  5) `Growing_area` → `Region`

Computing closure

- Closure F^+ of a set of functional dependencies F is used to refer to the set of all functional dependencies that can be derived from F.
- Example:

$\{A \rightarrow B, B \rightarrow C\}^+ = \{$	$A \rightarrow B, B \rightarrow C,$	
	$A \rightarrow C,$	F3
	$AB \rightarrow C, AB \rightarrow AC,$	F2
	$A \rightarrow A,$	F1
	$\ldots\}$	

Computing closure

- Closure X_F^+ of a set of attributes X with respect to a set of functional dependencies F is used to refer to the set of all attributes that functionally depend on X according to F.

  $$X_F^+ := \{ A \mid X \rightarrow A \in F^+ \}$$

  Because of rule F1, the following applies: $X \subseteq X_F^+$

- Examples:

  For $F=\{A \rightarrow B, B \rightarrow C\}$ with $R=\{A,B,C,D\}$ then

  $\{A\}_F^+ = \{A,B,C\}$
  $\{B\}_F^+ = \{B,C\}$
  $\{A,B\}_F^+ = \{A,B,C\}$
  $\{C\}_F^+ = \{C\}$
  $\{D\}_F^+ = \{D\}$

Membership problem

- Membership problem: given is a set of functional dependencies F. Can $A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$ be derived from F?

- Solution attempt 1:
  $A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$ can be derived from F if and only if $A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m \in F^+$
  - The problem with this: F^+ potentially contains exponentially many FDs, i.e. every algorithm for calculating F^+ has exponential runtime complexity. Therefore in practical terms, this solution attempt cannot be implemented.

- Solution attempt 2:
  $A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m$ can be derived from F if and only if $\{B_1, B_2, \ldots, B_m\}$ is a subset of the closure of $\{A_1, A_2, \ldots, A_n\}$ with regard to F then: $\{B_1, B_2, \ldots, B_m\} \subseteq \{A_1, A_2, \ldots, A_n\}_F^+$
  - An algorithm for calculating $\{A_1, A_2, \ldots, A_n\}_F^+$ in linear time does exist (CLOSURE algorithm)!

CLOSURE algorithm - idea

- Input: *F,* simplified set of functional dependencies
  $\{A_1, A_2, \ldots, A_n\}$, attributes
- Output: $\{A_1, A_2, \ldots, A_n\}_F^+$, closure of the attributes with regard to F
- Algorithm idea:
  1) $H = \{A_1, A_2, \ldots, A_n\}$
  2) For every FD $B_1, B_2, \ldots, B_m \to C$ in F:
     If all $B_1, B_2, \ldots, B_m$ are in H, but C is not in H, then add C to H.
  3) Repeat step 2 until no more attributes can be added to H.
  4) Now H contains the closure of $\{A_1, A_2, \ldots, A_n\}$ with regard to F.

F:	A→B	B→C	C→D	H={A}
Step 1				H={A,B}
Step 2				H={A,B,C}
Step 3				H={A,B,C,D}

D $\subseteq$ CLOSURE(F,{A}) $\Rightarrow$
A$\to$D can be derived from F.

CLOSURE and MEMBER algorithms

- Determination of X_F^+: the closure of X with regard to F.

Algorithm: **CLOSURE**(F, X):

```
    H := X
    repeat
        done = true
        forall FDs $Y \to C \in F$
            if $Y \subseteq H$ and $C \notin H$ then
                $H = H \cup \{C\}$, done = false
    until(done)
    return H
```

- Membership test

Algorithm: **MEMBER**(F, $X \to Y$): /* Test for $X \to Y \in F^+$ */

```
  return ($Y \subseteq$ CLOSURE(F, X))
```

Examples

- Example 2:
  - Relation
    `RWINES={Name, Colour, Vintage, Vineyard, Growing_area, Region}`
  - Given FDs:
    - `Name, Vintage, Vineyard` → `Growing_area, Colour`
    - `Growing_area` → `Region`
  - Question: does this FD apply?
    - `Name, Vintage, Vineyard, Colour` → `Region, Growing_area`

- Algorithm: does this apply?

`Region, Growing_area` ⊆ CLOSURE(F,{`Name, Vintage, Vineyard, Colour`})

F:	Name, Vintage, Vineyard → Growing_area, Colour	Growing_area → Region	H={Name, Vintage, Vineyard, Colour}
Step 1			H={Name, Vintage, Vineyard, Colour, Growing_area}
Step 1			H={Name, Vintage, Vineyard, Colour, Growing_area, Region}

Relationship between functional dependencies and keys

- A set of attributes $K \subseteq R$ is a **key** of R, if and only if
  1. $K \rightarrow R \in F$
  2. There is no real subset of $X \subset K$ with $X \rightarrow R$

  In words: the attributes in K uniquely determine all attributes of R and are minimal. Without (2) we speak of a **superkey**.

- Using MEMBER and CLOSURE, we can now calculate the following:
  1. K is a **superkey** of R if and only if
     $K_F^+ = R \Leftrightarrow$
     R = CLOSURE(F,K) $\Leftrightarrow$
     MEMBER(F,$K \rightarrow R$)
  2. K is a **key** of R if and only if additionally for all subsets $X \subset K$ the following also apply
     $X_F^+ \neq R \Leftrightarrow$
     $R \neq$ CLOSURE(F,X) $\Leftrightarrow$
     not(MEMBER(F,$X \rightarrow R$))

Key determination in practice (1)

- Generate all possible attribute combinations (candidates), and then check whether they are keys using the MEMBER tests as just shown
  - Checking is possible in linear time → OK
  - But exponentially many attribute combinations exist → not practicable

- Use of heuristics to reduce the number of candidates
  1. Simplify FDs F, through which the right sides become single-element sets and the trivial and duplicated FDs are removed.
  2. All attributes $S \subseteq R$ which do not appear in any FD must be in the key. For proof, see 3.
  3. All attributes $S \subseteq R$ which do not appear on any *right* side of an FD must be contained in the key.
     Proof: if S does not appear on any *(right)* side of the FD, then the following applies
     $$\forall K \subseteq \{R/S\}: S \notin CLOSURE(F, K),$$
     then K cannot be a key, because the following applies
     $$K_F^+ \neq R.$$
     Contradiction, S is in the key.
  4.

Key determination in practice (2)

4. Test whether the closure of the attribute set found contains all attributes
    1. Yes → only key

       Proof:

       Assumption: S is a key, none of the elements of S appear on the right side of an FD and there is no key $K \neq S$. Then the following applies

       $$\forall K \subseteq \{R/s\},\ s \in S:\ s \notin CLOSURE(F, K)$$

       then K cannot be a key, because the following applies

       $$K_F^+ \neq R.$$

       Contradiction, S is the only key, because the following also applies

       $$K_F^+ = R \text{ for } K = S \cup X \subseteq R,$$

       which would make K a superkey.
    2. No → try all other combinations successively: first add another attribute, then two, then three, etc. until all keys are found. Of course, only carry out the meaningful tests.

- The algorithm is still exponential, but in many cases it is practicable.

Key determination example 1

- Key determination for the relation
  `RWINES={Name,Colour,Vintage,Vineyard,Growing_area,Region}`
  - with the FDs F

```
Name, Vintage, Vineyard → Growing_area, Colour
Vineyard → Growing_area, Vineyard, Region
Growing_area → Region
```

1. Simplification (see example on simplification) results in

```
Name, Vintage, Vineyard  → Growing_area
Name, Vintage, Vineyard  → Colour
Vineyard → Growing_area
Vineyard → Region
Growing_area → Region
```

2. Non-existent attributes: none
3. Attributes which do not appear on the right: `Name, Vineyard, Vintage`
4. Computing the closure: $\{$`Name, Vineyard, Vintage`$\}_F^+$ = $\{$`Name, Vineyard, Vintage, Colour, Growing_area, Region`$\}$ = `RWINES`
   → is the only key!

➔ key for `RWINES`: {{`Name, Vineyard, Vintage`}}

Key determination example 2

- Key determination for `R(A,B,C,D)` with the FDs F

  $A, B, C \rightarrow D$

  $D \rightarrow A$

  1. Simplification results in: no change to the FDs
  2. Non-existent attributes: none
  3. Attributes which do not appear on the right: `B, C`
  4. Computing the closure: $\{B, C\}_F^+ = \{B, C\} \neq R$ → $\{B, C\}$ is NOT a key.
     Nevertheless, $\{B, C\}$ must be included in every key.
     Try successively:
     1. $\{B, C, A\}$: $\{B, C, A\}_F^+ = \{B, C, A, D\} = R$ → is the key
     2. $\{B, C, D\}$: $\{B, C, D\}_F^+ = \{B, C, D, A\} = R$ → is the key
     3. If the first two were both not keys, then we must try $\{B, C, A, D\}$. In this way, we know without further testing that this is a superkey, but not a key.

  ➔ Key of `R: {{A, B, C}, {B, C, D}}`

Equivalence of sets of FDs

- Two sets of FDs F and G are equivalent (written as $F \equiv G$) if they have the same closure: $F \equiv G \; := F^+ = G^+$
  - i.e. if the same FDs can be derived from them
  - e.g. $\{A \rightarrow B,\ B \rightarrow C\} \equiv \{A \rightarrow B,\ B \rightarrow C,\ A \rightarrow C\}$

- Closure F^+ of an FD set F is unique, but very big, therefore unwieldy.
  - The fewer FDs we have to work with, the more efficient the design process and the resulting database will be
  - For a given set of FDs, we are thus looking for the smallest possible equivalent set of FDs
  - SPLITTING(F) ≡ F. This is already easier, but it's possible to do even better.

Equivalence of sets of FDs, example

- Usually there are many different sets of FDs that are equivalent.
  - Example: `R=(ABC)`.
    The FDs F_1, F_2, F_3, F_4 are all equivalent ($F_1 \equiv F_2 \equiv F_3 \equiv F_4$)

    $F_1 = \{ \texttt{A} \rightarrow \texttt{B},\ \texttt{B} \rightarrow \texttt{C},\ \texttt{A} \rightarrow \texttt{C} \}$

    $F_2 = \{ \texttt{A} \rightarrow \texttt{B},\ \texttt{B} \rightarrow \texttt{C} \}$

    $F_3 = \{ \texttt{A} \rightarrow \texttt{B},\ \texttt{B} \rightarrow \texttt{C},\ \texttt{AB} \rightarrow \texttt{C} \}$

    $F_4 = \{ \texttt{A} \rightarrow \texttt{B},\ \texttt{B} \rightarrow \texttt{BC},\ \texttt{AC} \rightarrow \texttt{BC} \}$

    because they all span the same closure:

    $$F_1^+ = F_2^+ = F_3^+ = F_4^+ = \{ \texttt{A} \rightarrow \texttt{B},\ \texttt{B} \rightarrow \texttt{C},\ \texttt{A} \rightarrow \texttt{C},\ \texttt{A} \rightarrow \texttt{AB},\ \texttt{A} \rightarrow \texttt{BC},\ \texttt{A} \rightarrow \texttt{AC},\ \texttt{A} \rightarrow \texttt{ABC},\ \texttt{B} \rightarrow \texttt{BC},\ \texttt{AB} \rightarrow \texttt{AC},\ \texttt{AC} \rightarrow \texttt{BC},\ \texttt{AB} \rightarrow \texttt{C},\ \texttt{AB} \rightarrow \texttt{BC},\ \texttt{+ trivial FDs}\}$$

Canonical (minimal) cover

- The canonical cover (minimal cover) F_c of a set of FDs F meets the following conditions
  1) $F_c \equiv F$
  2) The left side of every FD $f \in F_c$ is unique.
     Because it is $\{A \rightarrow X, A \rightarrow Y\} \equiv \{A \rightarrow XY\}$, duplicate left sides can easily be removed.
  3) Neither the left nor the right side of every FD $X \rightarrow Y \in F_c$ contains superfluous attributes:
     (a) No left side can be shortened
     $$\forall A \in X: (F_c - (X \rightarrow Y)) \cup ((X\text{-}A) \rightarrow Y)) \not\equiv F_c$$
     (b) No right side can be shortened
     $$\forall B \in Y: (F_c - (X \rightarrow Y)) \cup (X \rightarrow (Y\text{-}B))) \not\equiv F_c$$

  - Example from the previous slide: F_2 is a canonical cover for F_1, F_3 , and F_4.

The cover algorithm is not deterministic, depends on the order in which the rules are processed.

Algorithm for computing the canonical cover

- Input: set of FDs F, output: canonical cover F_c

- Algorithm: **COVER**(F)
  1) Initialise $F_c = F$.
  2) Simplify F_c = SPLITTING(F_c):

     Replace every FD

     $A_1, A_2, \dots, A_n \rightarrow \boldsymbol{B_1, B_2, \dots, B_m} \in F_c$

     with the m FDs

     $A_1, A_2, \dots, A_n \rightarrow \boldsymbol{B_1}$

     $\dots$

     $A_1, A_2, \dots, A_n \rightarrow \boldsymbol{B_m}$

     and eliminate trivial FDs.

Algorithm for computing the canonical cover

3) Minimise left sides:
   for every FD
   $A_1, A_2, \ldots, A_n \rightarrow B \in F_c$ and every i = 1, …, n:
   if B can also be generated by F_c without an A_i
   $B \in$ CLOSURE(F_c,$\{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n\}$) or
   $B \in \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n\}_{F_c}^{+}$
   then remove A_i from the FD
   $F_c = F_c - \{A_1, A_2, \ldots, A_n \rightarrow B\} \cup \{A_1, \ldots, A_{i-1}, A_{i+1}, \ldots, A_n \rightarrow B\}$

4) Remove unnecessary FDs:
   for every FD $X \rightarrow B \in F_c$ if the FD can also be generated
   from F_c without this FD
   $B \in \{X\}_{(FC - (X \rightarrow B))}^{+}$,
   then remove this FD from F_c
   $F_c = F_c - \{X \rightarrow B\}$.

5) Summarising identical left sides (SPLITTING(F_c) reversed):
   replace all FDs $X \rightarrow \mathbf{B_1}, X \rightarrow \mathbf{B_2}, \ldots, X \rightarrow \mathbf{B_n} \in F_c$
   with the one FD $X \rightarrow \mathbf{B_1, B_2, \ldots, B_n}$

Example of computing a canonical cover

1. $F_4 = F_c = \{ \texttt{A} \rightarrow \texttt{B}, \texttt{B} \rightarrow \texttt{BC}, \texttt{AC} \rightarrow \texttt{BC} \}$
2. $SPLITTING(F_c) = \{ \texttt{A} \rightarrow \texttt{B}, \texttt{B} \rightarrow \texttt{C}, \texttt{AC} \rightarrow \texttt{B} \}$
3. Minimisation of left sides only necessary for $\texttt{AC} \rightarrow \texttt{B}$.
   (1) Test whether $\texttt{B} \in \{\texttt{C}\}_{F_c}^{+} = \{\texttt{C}\}$. No, so no change to F_c
   (2) Test whether $\texttt{B} \in \{\texttt{A}\}_{F_c}^{+} = \{\texttt{A}, \texttt{B}, \texttt{C}\}$. Yes, so
   $F_c = \{ \texttt{A} \rightarrow \texttt{B}, \texttt{B} \rightarrow \texttt{C}, \mathbf{A} \rightarrow \mathbf{B} \}$
4. Removal of unnecessary FDs:
   (1) for $\texttt{A} \rightarrow \texttt{B}$: test whether $\texttt{B} \in \{\texttt{A}\}_{\{B \rightarrow C\}}^{+} = \{\texttt{A}\}$. No, so no change.
   (1) for $\texttt{B} \rightarrow \texttt{C}$: test whether $\texttt{C} \in \{\texttt{B}\}_{\{A \rightarrow B\}}^{+} = \{\texttt{B}\}$. No, so no change.
5. Summarising left sides:
   There are no identical left sides.
   6. $F_c = \{ \texttt{A} \rightarrow \texttt{B}, \texttt{B} \rightarrow \texttt{C} \} = F_2$

Decomposition of relations to avoid redundancies

- If we put "too much" into a relational schema, we create redundancies. These lead to anomalies (update anomalies and deletion anomalies)

- Functional dependencies allow us to formalise our knowledge about the application domain. We can transform FDs. We can calculate the keys of a relation from FDs.

➔ Decomposition = splitting of relational schemas with the help of FDs so that the resulting relational schemas contain <u>no</u> redundancies anymore.

- Aim: complete avoidance of redundancies - but without simultaneously
  - losing semantic information (dependency preservation)
  - losing the possibility of reconstructing the relations (lossless join decomposition)

  (However, we will see that these goals are not always fully achievable.)

Normal forms and normalisation

- Normal forms
  - Specify properties of relational schemas
  - Prohibit certain combinations of functional dependencies in relations
- Normalisation
  - The process of converting a set of relational schemas that are not in a desired normal form into a new set of relational schemas that are all in the desired normal form

First normal form 1NF

- A relation R corresponds to the first normal form if
  1. Every attribute has an atomic range of values
  2. R is free of repeating groups

- Advantage regarding sortability and processing

Repeating groups

R:

Vehicle	Models
BMW 1 series	3ZB, 4ZB,..
BMW 3 series	R6ZB,R6ZD,..
Tata Estate	1ZG
Tata Sierra	4ZE

Non-atomic: Brand, Name

R:

Manufacturer	Name	Model
BMW	1 series	3ZB
BMW	1 series	4ZB
BMW	3 series	R6ZB
BMW	3 series	R6ZD
Tata	Estate	1ZG
Tata	Sierra	4ZE

Second normal form 2NF

- A relation R corresponds to the second normal form if
  1. R corresponds to the first normal form
  2. Every attribute that is not part of a key depends on the whole key, and not just on a real subset.

- Advantage: monothematic tables

R:

Brand	Name	Norm	HQ
BMW	1 series	Euro5A	Munich
BMW	3 series	Euro4	Munich
Tata	Estate	Euro2	Mumbai
Tata	Sierra	Euro1	Mumbai

Clearly: the redundancy in the company headquarters (HQ) is not advantageous: it's a waste of storage space and can result in update anomalies and deletion anomalies.

But: how do we check the second criterion?

Second normal form 2NF

- A relation R corresponds to the second normal form if
  1. R corresponds to the first normal form
  2. Every attribute that is not part of a key depends on the whole key, and not just on a real subset.

- Advantage: monothematic tables

R:

Brand	Name	Norm	HQ
BMW	1 series	Euro5A	Munich
BMW	3 series	Euro4	Munich
Tata	Estate	Euro2	Mumbai
Tata	Sierra	Euro1	Mumbai

⇩ ✓

R1:

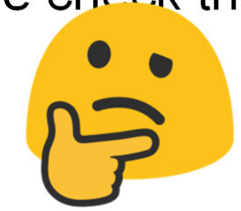
Brand	Name	Norm
BMW	1 series	Euro5A
BMW	3 series	Euro4
Tata	Estate	Euro2
Tata	Sierra	Euro1

R2:

Brand	HQ
BMW	Munich
Tata	Mumbai

Clearly: the redundancy in the company headquarters (HQ) is not advantageous: it's a waste of storage space and can result in update anomalies and deletion anomalies.

But: how do we check the second criterion?



Second normal form 2NF

R:

Brand	Name	Norm	HQ
BMW	1 series	Euro5A	Munich
BMW	3 series	Euro4	Munich
Tata	Estate	Euro2	Mumbai
Tata	Sierra	Euro1	Mumbai

F: $\{Brand,Name\} \rightarrow \{Norm, HQ\}$
$Brand \rightarrow HQ$
$\{Brand,Name\} \rightarrow Norm$

1. Keys via heuristics:
i) Are there attributes that are not in any FD? No (otherwise they would be in the key)
ii) All attributes that are not on the right side are in the {Brand,Name} key

2. Test key
i) $\{Brand,Name\}_F^+ = R$
ii) So {Brand,Name} is the only key
iii) Just for fun: also can't be shortened?
$\{Brand\}_F^+ = \{Brand,HQ\} \neq R$
$\{Name\}_F^+ = \{Name\} \neq R$, works!

3. F_C
i) SPLIT(F)= $\{Brand,Name\} \rightarrow Norm$
$\{Brand,Name\} \rightarrow HQ$
$Brand \rightarrow HQ$
ii) Shorten $\{Brand,Name\} \rightarrow Norm$
$Norm \notin \{Name\}_F^+$
$Norm \notin \{Brand\}_F^+$
iii) Shorten $\{Brand,Name\} \rightarrow HQ$
$HQ \notin \{Name\}_F^+$
$HQ \in \{Brand\}_F^+$
$\Rightarrow F_c = F \setminus \{Brand,Name\} \rightarrow HQ$
$\cup\ Brand \rightarrow HQ$
iv) $F_C =$ $\{Brand,Name\} \rightarrow Norm$
$Brand \rightarrow HQ$ ⚡

Violates 2NF constraint: every attribute that is not part of a key (HQ) depends on the whole key, and not just on a real subset (Brand).

Third normal form 3NF

- A relation R corresponds to the third normal form if
  1. R corresponds to the second normal form
  2. No non-key attribute is transitively dependent on a key. In other words: every left side is a superkey or the right side is prime.

- Advantage: monothematic tables

R:

Brand	Name	Norm	CO
BMW	1 series	Euro5A	1000
BMW	3 series	Euro4	1000
Tata	Estate	Euro2	2200
Tata	Sierra	Euro1	2720

F = {Brand,Name}→Norm
Norm→CO

CO is transitively dependent on a key ⚡

⇩ ✓

R1:

Brand	Name	Norm
BMW	1 series	Euro5A
BMW	3 series	Euro4
Tata	Estate	Euro2
Tata	Sierra	Euro1

R2:

Norm	CO
Euro5A	1000
Euro4	1000
Euro3	2300
Euro2	2200
Euro1	2720

Third normal form 3NF

R:

Brand	Name	Norm	CO
BMW	1 series	Euro5A	1000
BMW	3 series	Euro4	1000
Tata	Estate	Euro2	2200
Tata	Sierra	Euro1	2720

F: $\{Brand,Name\} \rightarrow Norm$
$Norm \rightarrow CO$

1. Keys via heuristics:
i) Are there attributes that are not in any FD? No (otherwise they would be in the key)
ii) All attributes that are not on the right side are in the {Brand,Name} key

2. Test key
i) $\{Brand,Name\}_F^+ = R$
ii) So {Brand,Name} is the only key
iii) Just for fun: also can't be shortened?
$\{Brand\}_F^+ = \{Brand,HQ\} \neq R$
$\{Name\}_F^+ = \{Name\} \neq R$, works!

3. F_C
SPLIT(F)= $\{Brand,Name\} \rightarrow Norm$
$Norm \rightarrow CO$

i) Shorten $\{Brand,Name\} \rightarrow Norm$
$Norm \notin \{Name\}_F^+$
$Norm \notin \{Brand\}_F^+$
ii) $F_C = \{Brand,Name\} \rightarrow Norm$
$Norm \rightarrow CO$

Norm is not a superkey and
CO is also not prime → violates 3NF

Alternative constraint for 3NF

- Third normal form (3NF)
  - Relation R with a set of simplified FDs F is in 3NF if and only if the left side of every FD is a superkey of R or the right side is prime.
  - Formally: relation R with simplified set of FDs F is in 3NF if and only if:

    $\forall X \rightarrow A \in F^+$ the following applies: $X \rightarrow (A$ trivial $\vee) X$ is a superkey of $R \vee$ A is prime
  - An attribute is called prime if it is part of a key.
  - Every relational schema in BCNF is automatically also in 3NF

How do we practically determine whether a relation is in 3NF?

- Given: a relation R with an FD set F

- Compute the canonical cover F_c from F.
  R is in 3NF if and only if the left sides of all FDs in F_c are superkeys or the right side is prime.

- The problem with this: to test if an attribute is prime (i.e. part of a key), we need to know all the keys of R. However, as we have seen, computing these takes exponential time in the worst case!
  - Nevertheless, in practice this is usually possible.

Decomposition: Normalisation of relations in 3NF

1) Given: set of relational schema Z + simplified set of FDs
   - In the simplest case of only one relation R is $Z = \{ R \}$.

2) As long as there is still a relational schema $S \in Z$ that is not in 3NF:
   - Find a non-trivial FD $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ valid for S that violates the 3NF (i.e. $\{A_1, \ldots, A_n\}$ is not a superkey of S or at least one B_i is not prime)
   - Compute $S_1 = \{A_1, \ldots, A_n\}^+$ and $S_2 = S - S_1 \cup \{A_1, \ldots, A_n\}$
   - Remove S from Z and insert S_1 and S_2 into Z, i.e. $Z = Z - \{S\} \cup \{S_1, S_2\}$
   - Assign the FDs to the (newly-created) relations

3) Z is now a 3NF compliant decomposition of the original schema

- Comments
  - S_1 is respectively the maximum attribute set functionally determined by $\{A_1, \ldots, A_n\}$ (which of course contains both $\{A_1, \ldots, A_n\}$ and $\{B_1, \ldots, B_m\}$)
  - S_2 is respectively the set of all other attributes combined with $\{A_1, \ldots, A_n\}$
  - This algorithm is also not deterministic, so depending on which rule is used first, there might be no meaningful result achieved.

3NF decomposition

SPLIT(F)= $\{Brand,Name\} \rightarrow Norm$
$Norm \rightarrow CO$

Norm is not a superkey and
CO is also not prime → violates 3NF

$S_1 = \{A_1, \ldots, A_n\}^+$ and $S_2 = S - S_1 \cup \{A_1, \ldots, A_n\}$
$S_1 = \{Norm\}^+$ and $S_2 = S - \{Norm, CO\} \cup \{Norm\}$

R:

Brand	Name	Norm	CO
BMW	1 series	Euro5A	1000
BMW	3 series	Euro4	1000
Tata	Estate	Euro2	2200
Tata	Sierra	Euro1	2720

⇨ ✓

R1:

Brand	Name	Norm
BMW	1 series	Euro5A
BMW	3 series	Euro4
Tata	Estate	Euro2
Tata	Sierra	Euro1

R2:

Norm	CO
Euro5A	1000
Euro4	1000
Euro3	2300
Euro2	2200
Euro1	2720

Boyce-Codd normal form (BCNF)

- A relation R corresponds to the Boyce-Codd normal form if
  1. R corresponds to the third normal form
  2. Every attribute set that other attributes functionally depend on is a superkey.

- Advantage: monothematic tables

R:

Brand	Name	Norm	Scandal	Severity
ACME	13 serie	Euro5A	No	None
ACME	Benny	Euro4	Yes	Low
Umbrella	Van	Euro2	No	None
Umbrella	Truck	Euro1	Yes	High

F= {Brand,Name}→Norm
{Brand,Name,Norm}→Scandal
{Norm,Scandal}→Severity
$\{Brand,Name\}_F^+$=R is the key.

Scandal and Severity are not transitively dependent on this key, R is 3NF, 2NF, 1N
But: Severity depends on
{Norm,Scandal}
and that is not a (super)key.

R1:

Brand	Name	Norm	Scanda
ACME	13 serie	Euro5A	No
ACME	Benny	Euro4	Yes
Umbrella	Van	Euro2	No
Umbrella	Truck	Euro1	Yes

R2:

Norm	Scandal	Severity

How do we practically determine whether a relation is in BCNF?

- Given: a relation R with an FD set F
- Compute the canonical cover F_c from F.
  R is in BCNF if and only if the left sides of all FDs in F_c are superkeys.
- Often a simpler alternative to show that R is <u>not</u> in BCNF is to give a counter-example!
  - Example: relation `RWINES` with the FDs
    - `Name, Vintage, Vineyard` → `Growing_area, Colour`
      `Vineyard` → `Growing_area, Vineyard, Region`
      `Growing_area` → `Region`
    - The keys of `RWINES` are (as already determined): {{`Name,Vineyard,Vintage`}}
    - `RWINES` is <u>not</u> in BCNF, for example because the left side of the FD `Growing_area` → `Region` (so {`Growing_area`}) is not a superkey.
- All relations with only 2 attributes are in BCNF.
  - Proof: see CompleteBook

Example of BCNF determination

$F = \{Brand, Name\} \rightarrow Norm$
$\{Brand, Name, Norm\} \rightarrow Scandal$
$\{Norm, Scandal\} \rightarrow Severity$

$F_c = SPLIT(F) = F$

$Norm \notin CLOSURE(F, Brand) = \{Brand\}$
$Norm \notin CLOSURE(F, Name) = \{Name\}$
$Scandal \in CLOSURE(F, \{Brand, Name\}) = \{Brand, Name, Norm, Scandal, Severity\}$
$\Rightarrow F_c = F \setminus \{Brand, Name, Norm\} \rightarrow Scandal$
$\cup \{Brand, Name\} \rightarrow Scandal$
$Scandal \notin CLOSURE(F, \{Brand, Norm\}) = \{Brand, Norm\}$
$Scandal \notin CLOSURE(F, \{Name, Norm\}) = \{Name, Norm\}$
$Severity \notin CLOSURE(F, Norm) = \{Norm\}$
$Severity \notin CLOSURE(F, Scandal) = \{Scandal\}$

$F_c = \{Brand, Name\} \rightarrow \{Norm, Scandal\}$
$\{Norm, Scandal\} \rightarrow Severity$

Compute the canonical cover Fc from F.
R is in BCNF if and only if the left sides of all FDs in Fc are superkeys.

Decomposition: Normalisation of relations in BCNF (1)

1) Given: set of relational schema Z and a simplified set of FDs

2) As long as there is still a relational schema $R \in Z$ that is not in BCNF:

- Find a non-trivial FD valid for R $A_1, \dots, A_n \rightarrow B_1, \dots, B_m$, which violates the BCNF (i.e. $\{A_1, \dots, A_n\}$ is not a superkey of S)
- Compute $S_1 = \{A_1, \dots, A_n\}^+$ and $S_2 = (R - S_1) \cup \{A_1, \dots, A_n\}$
- Remove R from Z and insert S_1 and S_2 into Z , i.e. $Z = Z - \{S\} \cup \{S_1, S_2\}$
- Assign the FDs to the (newly-created) relations

F_c = {Brand, Name}→Norm
{Brand, Name}→Scandal
{Norm, Scandal}→Severity ⚡

S_1 = $\{\text{Norm,Scandal}\}_F^+$
= {Norm,Scandal,Severity}

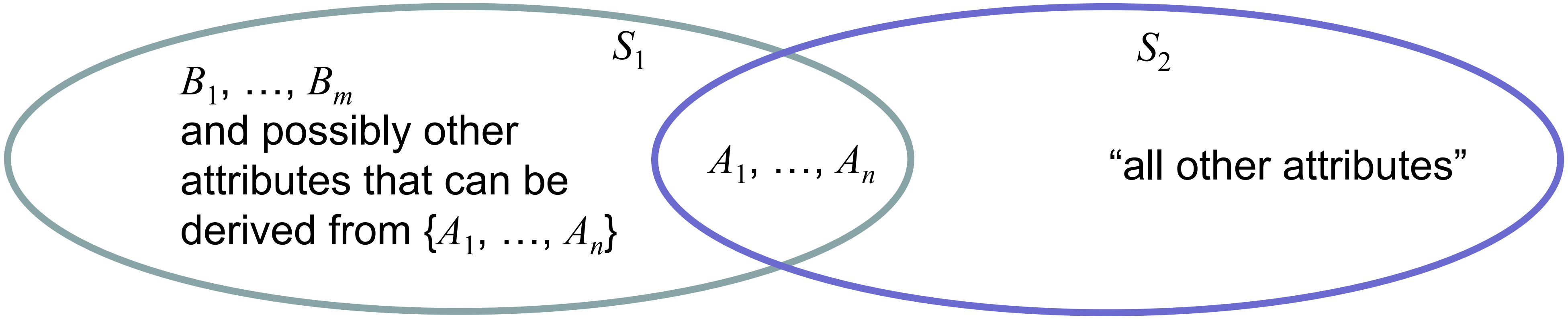
S_2 = R-S_1 ∪ {Norm,Scandal}
={Brand,Name,Norm,Scandal}

3) Z is now a BCNF compliant decomposition of the original schema

Decomposition: Normalisation of relations in BCNF (2)

- We can distinguish between three attribute sets when it comes to decomposition:
  1) $\{A_1, \ldots, A_n\}$: the left side of the FD that violates the BCNF
  2) $\{A_1, \ldots, A_n\}^+$: the closure of the left side of the FD that violates the BCNF. This of course contains the complete right side $\{B_1, \ldots, B_m\}$ of this FD, and also the complete left side (both can be derived trivially by means of the FD from $\{A_1, \ldots, A_n\}$), as well as potentially further attributes that can be derived from $\{A_1, \ldots, A_n\}$ by means of other FDs
  3) All other attributes.

- Visual representation of the decomposition of S


S_1
S_2
$B_1, \ldots, B_m$ and possibly other attributes that can be derived from $\{A_1, \ldots, A_n\}$
$A_1, \ldots, A_n$
"all other attributes"

Example of decomposition in BCNF

- Consider
  - Relation `PRODUCER = {Vineyard, Growing_area, Region}` with the FDs
    - `FD1: Vineyard → Growing_area`
    - `FD2: Growing_area → Region`
  - Determination of the keys of R results in {{`Vineyard`}}

- Decomposition algorithm
  1) Z = {`PRODUCER`}
  2) - `PRODUCER` is not in BCNF because of the FD `Growing_area → Region`, because {`Growing_area`} is not a superkey.
     - `PRODUCER1` = {`Growing_area`}$^+$ = {`Growing_area,Region`}
     - `PRODUCER2` = `PRODUCER` – `PRODUCER1` ∪ {`Growing_area`} =
       = {`Vineyard,Growing_area,Region`} – {`Growing_area,Region`} ∪ {`Growing_area`}
       = {`Vineyard, Growing_area`}
     - `FD1` is allocated to `PRODUCER2` (only there are all attributes from `FD1` present) and `FD2` is allocated to `PRODUCER1` (only there are all attributes from `FD2` present)
     - Z = {`PRODUCER1`, `PRODUCER2`}, all are in BCNF
  3) Thus, {`PRODUCER1`, `PRODUCER2`} is a BCNF-compliant decomposition

Requirements for decomposition / transformation

- We would generally like to have three important properties for the decomposition (transformation) of a relational schema:

1) **Elimination of the anomalies**
   - No more anomalies should occur in the resulting relational schemas

2) **Lossless join decomposition**
   - Precisely those tuples (application data) of the original relation should be able to be derived again from the tuples of the decomposed relational schemas

3) **Dependency preservation**
   - The functional dependencies that can be derived from the keys of the decomposed relations should be equivalent to the original FDs

Lossless join decomposition

- To meet the criteria of the normal forms, relational schemas are decomposed into smaller relational schemas

- To restrict ourselves to "meaningful" decomposition, the requirement is that the original relation can be recovered again from the decomposed relations by means of natural join
  - This must apply to every database that meets the FDs!

➔ Lossless join decomposition

Lossless join decomposition – example 1

- Given

  R = ABC with
  $F = \{A \rightarrow B,\ C \rightarrow B\}$

- Decomposition into

  $R_1 = AB$ and $R_2 = BC$

- Is ***not*** a lossless join decomposition:

A	B	C
1	2	3
4	2	5

A	B
1	2
4	2

B	C
2	3
2	5

A	B	C
1	2	3
4	2	5
1	2	5
4	2	3

Lossless join decomposition – example 2

- Given

  R = ABC with
  $F = \{A \rightarrow B,\ B \rightarrow C\}$

- Decomposition into

  $R_1 = AB$ and $R_2 = BC$

- Is a lossless join decomposition:

A	B	C
1	2	3
4	2	3

A	B
1	2
4	2

B	C
2	3

A	B	C
1	2	3
4	2	3

Lossless join decomposition - formally

- Formally: the decomposition of an attribute set

  X *into* $X_1, \ldots, X_p$ with $X = \bigcup^{p}_{i=1} X_i$

  is a lossless join decomposition ($\pi \; \triangleright\!\triangleleft$ lossless) with regard to a set of FDs F if and only if the following applies to all relations $r(X)$ that fulfil FDs F:

  $$\pi_{X_1}(r) \triangleright\!\triangleleft \; \ldots \; \triangleright\!\triangleleft \; \pi_{X_p}(r) = r$$

  - Comment: we can't lose any tuples due to the projection-join sequence, but we could get additional tuples. So why is it called "lossless"? Because we have lost the information about which tuples were originally there!
  - Decomposition is “lossy” if R1 ⋈ R2 ⊃ R (example 1)
  - Decomposition is “lossless” if R1 ⋈ R2 = R (example 2)

Lossless join decomposition - formally

- Simple criterion for lossless join decomposition when decomposing into two relational schemas: decomposition of

  X into X_1 and X_2

  is lossless join decomposition with regard to F if the intersection of the attributes is a superkey for at least one of the resulting relations:

$(X_1 \cap X_2) \rightarrow X_1 \in F^+$ or
$(X_1 \cap X_2) \rightarrow X_2 \in F^+$

F_c = {Brand, Name}→Norm
{Brand, Name}→Scandal
⚡{Norm, Scandal}→Severity

S_1 = $\{Norm,Scandal\}_F^+$
= {Norm,Scandal,Severity}

$S_2 = R-S_1 \cup$ {Norm,Scandal}
={Brand,Name,Norm,Scandal}

$(S_1 \cap S_2)$ = {Norm,Scandal}
$\{Norm,Scandal\}_F^+=S_1$
Superkey, always chosen as such

BCNF decomposition is always lossless join decomposition!

Dependency preservation

- The starting point was a relational schema R with a set of FDs F.
- After the decomposition (transformation), R no longer exists, but instead a set of relational schemas $\{R_1, \ldots, R_n\}$, whereby each R_i contains some, but not necessarily all, of the attributes of R.
- Therefore, there may be FDs in F that cannot be completely checked in any R_i because they contain attributes that do not occur together in any R_i
  - We could of course compute R by joins from the R_i and then check the FDs, but that is far too much effort
- We define the key dependencies F_{R_i} on R_i as the set of all FDs from F^+ that only contain attributes from R_i and whose left side is a key for R_i.
- Decomposition is dependency preserving if and only if

  $F \equiv F_{R_1} \cup \ldots \cup F_{R_n}$ (or $F^+ = (F_{R_1} \cup \ldots \cup F_{R_n})^+$)

Dependency preservation – example

- Decomposition of the relation
  - `PRODUCER = {Vineyard, Growing_area, Region}` with
    - `Vineyard → Growing_area`
    - `Growing_area → Region`

A	B	C
1	2	3
4	2	3

- Into the relations
  - `E1` = {`Growing_area, Region`} with keys {{`Growing_area`}} and
  - `E2` = {`Vineyard, Growing_area`} with keys {{`Vineyard`}}

A	B
1	2
4	2

B	C
2	3

- Therefore
  - F_{E1}= {`Growing_area → Region`} $\cup$ trivial FDs
  - F_{E2}= {`Vineyard → Growing_area`} $\cup$ trivial FDs

- The following obviously applies $F^+ = (F_{E1} \cup F_{E2})^+$ since $F = F_{E1} \cup F_{E2}$

- This BCNF decomposition is therefore dependency preserving.

Non-dependency preserving decomposition in BCNF – example (1)

Relation `POSTCODE_DIRECTORY` with attributes `Street, Place, State, Postcode`

- Places are uniquely identified by name and state
- The postcode does not change within a street
- Postcode areas do not cross place boundaries and places do not cross state boundaries
- This results in the FDs F:
  - `Postcode` $\rightarrow$ `Place, State`
  - `Street, Place, State` $\rightarrow$ `Postcode`
- This results in the following as the key for the relation `POSTCODE_DIRECTORY`:
  - {{`Street, Place, State`}, {`Street, Postcode`}}

Is `POSTCODE_DIRECTORY` in BCNF?

- Are all the left sides of F_C superkeys?
  - F_C=SPLIT(F)={Postcode→Place, Postcode→State, {Street, Place, State}→Postcode}
  - Postcode $\notin$ $\{Street,Place\}_F^+$, Postcode $\notin$ $\{Street,State\}_F^+$, PLZ $\notin$ $\{State,Place\}_F^+$,
  - F_C=F.
- No, Postcode is not a superkey. Therefore `Postcode` $\rightarrow$ `Place, State` violates BCNF

Source: Kemper/Eickler: Datenbanksysteme

Non-dependency preserving decomposition in BCNF – example (1)

- Application of the decomposition algorithm for FD
  `Postcode → Place, State`
  - `S1 = {Postcode}`$_F^+$`={Postcode,Place,State},`
  - `S2 = R-S1 ∪ Postcode = {Street,Postcode}`
  - This results in the relational schemas
    `PLACES` = {`Postcode, Place, State`} and
    `STREETS` = {`Postcode, Street`}

- Anomalies have been eliminated, the new schema is a lossless join decomposition (according to the simple criterion: intersection is {`Postcode`} and this is a key for `PLACES` BUT the FD

  `Street, Place, State → Postcode`

  is no longer assigned to any relation. This is not dependency preserving!

Source: Kemper/Eickler: Datenbanksysteme

Non-dependency preserving decomposition in BCNF – example (2)

Problem with the lost FD Street, Place, State → Postcode:

POSTCODE_DIRECTORY

Place	State	Street	Postcode
Frankfurt	Hesse	Göthestraße	60313
Frankfurt	Hesse	Galgenstraße	60437
Frankfurt	Brandenburg	Göthestraße	15234

$\pi_{Postcode,Street}$

STREETS

Postcode	Street
60313	Göthestraße
60437	Galgenstraße
15234	Göthestraße

insert:

15235	**Göthestraße**

$\pi_{Place,State,Postcode}$

PLACES

Place	State	Postcode
Frankfurt	Hesse	60313
Frankfurt	Hesse	60437
Frankfurt	Brandenburg	15234

insert:

Frankfurt	**Brandenburg**	**15235**

PLACES ⋈ STREETS

Place	State	Street	Postcode
Frankfurt	Hesse	Göthestraße	60313
Frankfurt	Hesse	Galgenstraße	60437
Frankfurt	Brandenburg	Göthestraße	15234
Frankfurt	Brandenburg	Göthestraße	15235

to FD!

Source: Kemper/Eickler: Datenbanksysteme

Transformation requirements and decomposition in BCNF

- Transformation requirements and the BCNF decomposition
  1) Elimination of the anomalies
     - Fulfilled: no more redundancy-generating FDs in the decomposed relations.
  2) Lossless join decomposition
     - Fulfilled: decomposition into S1 and S2 is precisely chosen so that {A1, …, An} is the intersection of the two and at the same time a key of S1
       ➔ The simple constraint for lossless join decomposition is fulfilled.
  3) Dependency preservation
     - Not always fulfilled: we have just seen a counter-example.

- However, practical experience shows: BCNF decomposition is almost always dependency preserving!
  - However, if not: generated DB schema generally not acceptable, all FDs must be guaranteed by the generated DB schema.
  - Question: is there another normal form that meets all three requirements?
  - Answer: no, but there is the 3NF, which meets 2) and 3) but not 1).

Dependency preservation in practice

How do we determine in practice whether a decomposition is dependency preserving?

- Given
  - a relation R with an FD set F.
  - a decomposition of R into n relations $R_1, \ldots, R_n$ each with keys $\{K_{i1,} \ldots, K_{i,m\_i}\}$.
- G = $\{K_{ij} \rightarrow R_i\}$ is the set of all FDs generated from the keys.
- The decomposition is dependency preserving if and only if we can derive every FD $A_1, \ldots, A_n \rightarrow B \in F$ from the FDs G.
  - Or specifically: if the closure $\{A_1, \ldots, A_n\}_G^+$ contains the attribute B (member test)

The normal forms and their constraints

1NF	Every attribute has an atomic range of values R is free of repeating groups
2NF	Every attribute that is not part of a key depends on the whole key, and not just on a real subset.
3NF	No non-key attribute is transitively dependent on a key.
BCNF	Every attribute set that other attributes functionally depend on is a superkey.

Every non-key attribute must provide a fact about the key (1NF), the whole key (2NF), and nothing but the key (3NF), so help me Codd (and Boyce for all (BCNF)).

Comparison between 3NF and BCNF decomposition

- The 3NF decomposition algorithm which we have learned so far runs in exponential time complexity, since it requires the set of all keys to be determined → in practice this is hardly usable.
- The version derived from this for decomposition in BCNF runs in polynomial time, but is not always dependency preserving.
- So now let's look at a better version: synthesis process.

Synthesis process - idea

- Principle:
  Synthesis transforms the original FD set F into the resulting set of key dependencies G in such a way that $F \equiv G$ applies
  - Dependency preservation is anchored in the process
  - Lossless join decomposition is also anchored in the process
  - 3NF is always achieved (often even BCNF – but this must be checked in every case)

- Time complexity: quadratic

Synthesis process - algorithm

- Synthesis process: procedure
  - Given: relational schema R with FDs F.
  - Wanted: lossless join & dependency preserving decomposition into $R_1, \ldots, R_n$, whereby all R_i are in 3NF

- Algorithm: **`SYNTHESIZE`**(R, F):
  1) Compute the canonical cover F_c of F
  2) For every left side of an $A_1, \ldots, A_n$ of an FD in F_c, generate a relation with the attributes $\{A_1, \ldots, A_n\} \cup \{ B \mid A_1, \ldots, A_n \to B \in F_c\}$
  3) If none of the generated relations is a superkey of R, generate another relation which consists of the attributes of a key of R.
  4) Remove any relation that is completely contained in another relation.

Example synthesis process (1)

Relation `POSTCODE_DIRECTORY` with attributes `Street, Place, State, Postcode`

- $F_C = F$:
  - `Postcode → Place, State`
  - `Street, Place, State → Postcode`

Is `POSTCODE_DIRECTORY` in BCNF?

- Are all the left sides of F_C superkeys?
  - F_C=SPLIT(F)={Postcode→Place, Postcode→State, {Street, Place, State}→Postcode}
  - Postcode $\notin$ {Street,Place}$_F^+$, Postcode $\notin$ {Street,State}$_F^+$, PLZ $\notin$ {State,Place}$_F^+$,
  - F_C=F.
- No, Postcode is not a superkey. Therefore `Postcode → Place, State` violates BCNF

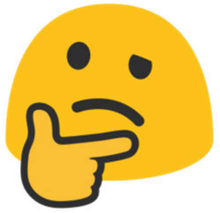
Synthesis process results in decomposition:

- R1={Postcode,Place,State}
- R2=R={Street, Place, State, Postcode} (pointless, since no decomposition)

Example synthesis process (2)

R:

Brand	Name	ssion_stand	HQ
BMW	1 series	Euro5A	Munich
BMW	3 series	Euro4	Munich
Tata	Estate	Euro2	Mumbai
Tata	Sierra	Euro1	Mumbai



R was not 3NF

i) F_C= {Brand,Name}→Norm
Brand→HQ

ii) R1:

Brand	Name	ssion_stand
BMW	1 series	Euro5A
BMW	3 series	Euro4
Tata	Estate	Euro2
Tata	Sierra	Euro1

R2:

Brand	HQ
BMW	Munich
BMW	Munich
Tata	Mumbai
Tata	Mumbai

iii) {Brand, Name, Norm} is a superkey of R.
iv) There is no relation that is completely contained in another relation. → Finished.

Example synthesis process (3)

- Example
  - Relation R with attributes ABCE
  - FD set F = {A → B, AB → C, A → C, B → A, C → E }

1) Canonical cover: F_c = {A → B, B → C, B → A, C → E }

2) Relations generated: {AB, BCA, CE}

3) {AB} is a superkey for ABCE (since $\{AB\}^+ = \{ABCE\}$) so no further relation necessary

4) Elimination of the first generated relation, since it's completely contained in the second.

➔ Result of the synthesis in 3NF: {ABC, CE}
(Comment: test for BCNF by means of CLOSURE algorithm: schema is also in BCNF)

Synthesis process - achieving lossless join decomposition

- So far: achieve lossless join decomposition by step 3 in the algorithm
  - Test whether a relation is a superkey: polynomial
  - However, if not: determining the key is exponential!
  - Therefore, the synthesis process in this form is exponential!

- Improvement: achieving lossless join decomposition with a simple "trick":
  - Instead of step 3
  - Expand the original FD set F with $R \rightarrow \delta$ whereby δ is a dummy attribute
  - δ is removed from all relations and FDs after synthesis

- Example: R=ABCE, F={A $\rightarrow$ B, C $\rightarrow$ E }
  - Synthesis result {AB, CE} is not lossless join decomposition because neither AB nor CE are superkeys (the only key is AC)
  - Dummy FD ABCE $\rightarrow \delta$ in F becomes AC $\rightarrow \delta$ in $F_{c.}$
  - And provides a third relational schema {AC}, so that lossless join decomposition synthesis in 3NF is ensured by {AB, CE, AC}

Summary

- Functional dependencies
- Keys
- Heuristics for key determination
- Member test using CLOSURE
- Equality of FDs using COVER
- 1NF, 2NF, 3NF, BCNF to avoid anomalies
- Decomposition
- Synthesis process

Decomposition algorithms and properties

Algorithm	Lossless join decomposition	Dependency preservation	Solution	Time complexity	Advantages/ disadvantages
Decomposition 3NF	Yes	Not always	3NF if dependency preserving	Exp.	Exponential runtime, few relations
Decomposition BCNF	Yes	Not always	BCNF if dependency preserving	Poly.	Often works, few relations
Synthesis	Not always	Yes	At least 3NF if lossless join decomposition	Exp.	Many relations
Advanced synthesis	Yes	Yes	At least 3NF, often BCNF	Poly.	Always works, many relations

Additional slide (not relevant for the examination)

- Why does determining all real keys run in exponential time complexity?
- In the worst case, we must compute the cover of all subsets of our attribute set. The number of all subsets of an n-element set is referred to as the *Bell number*. The following applies to the growth of these Bell numbers:

$$B_n < \left(\frac{0.792n}{\ln(n+1)}\right)^n, \qquad n \in \mathbf{N}.$$

- *Improved Bounds on Bell Numbers and on Moments of Sums of Random Variables, Daniel Berend, Tamir Tassa*

  S.L. Osborn. Normal Forms for Relational Databases.