Relational Database Systems 1

Christoph Lofi
Simon Barthel
Institut für Informationssysteme
Technische Universität Braunschweig
www.ifis.cs.tu-bs.de
• Up to now, we have learned ...
  – …how to model schemas from a conceptual point of view
  – … how the relational model works.
  – … how it is implemented in current RDBMS.
  – … how to create relational databases (SQL DDL).
  – … how to define constraints (SQL DDL).
  – … how to query relational databases.
  – … how to insert, delete, and update data (SQL DML).

• What’s missing?
  – How to create a good database design?
  – By the way: What is a good database design?
• Which table design is better?

A

<table>
<thead>
<tr>
<th>hero_id</th>
<th>team_id</th>
<th>hero_name</th>
<th>team_name</th>
<th>join_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Thor</td>
<td>The Avengers</td>
<td>1963</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Mister Fantastic</td>
<td>Fantastic Four</td>
<td>1961</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Iron Man</td>
<td>The Avengers</td>
<td>1963</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Hulk</td>
<td>The Avengers</td>
<td>1963</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Captain America</td>
<td>The Avengers</td>
<td>1964</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Invisible Girl</td>
<td>Fantastic Four</td>
<td>1961</td>
</tr>
</tbody>
</table>

B

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Thor</td>
</tr>
<tr>
<td>2</td>
<td>Mister Fantastic</td>
</tr>
<tr>
<td>3</td>
<td>Iron Man</td>
</tr>
<tr>
<td>4</td>
<td>Hulk</td>
</tr>
<tr>
<td>5</td>
<td>Captain America</td>
</tr>
<tr>
<td>6</td>
<td>Invisible Girl</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>team_id</th>
<th>team_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Avengers</td>
</tr>
<tr>
<td>2</td>
<td>Fantastic Four</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hero_id</th>
<th>team_id</th>
<th>join_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1961</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1964</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1961</td>
</tr>
</tbody>
</table>
• What’s wrong with design A?
  – **redundancy**: the team names are stored several times
  – **inferior expressiveness**: we cannot nicely represent heroes that currently have no team.
  – **modification anomalies**: (see next slide)

<table>
<thead>
<tr>
<th>hero_id</th>
<th>team_id</th>
<th>hero_name</th>
<th>team_name</th>
<th>join_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Thor</td>
<td>The Avengers</td>
<td>1963</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Mister Fantastic</td>
<td>Fantastic Four</td>
<td>1961</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Iron Man</td>
<td>The Avengers</td>
<td>1963</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>Hulk</td>
<td>The Avengers</td>
<td>1963</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>Captain America</td>
<td>The Avengers</td>
<td>1964</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>Invisible Girl</td>
<td>Fantastic Four</td>
<td>1961</td>
</tr>
</tbody>
</table>
There are three kinds of modification anomalies

- **insertion anomalies**
  - how do you add heroes that currently have no team?
  - how do you (consistently!) add new tuples?

- **deletion anomalies**
  - deleting Mister Fantastic and Invisible Girl also deletes all information about the Fantastic Four

- **update anomalies**
  - renaming a team requires updating several tuples (due to redundancy)
In general, **good relational database designs** have the following properties

- redundancy is minimized
  - i.e. no information is represented several times!
  - logically distinct information is placed in distinct relation schemes

- modification anomalies are prevented by design
  - i.e. by using keys and foreign keys, not by enforcing an excessive amount of (hard to check) constraints!

- in practice, **good designs should also match the characteristics of the used RDBMS**
  - enable efficient query processing
  - ....this, however, might in some cases mean that redundancy is beneficial
    - It’s quite tricky to find the proper balance between different optimization goals

- In essence, it’s all about splitting up tables ...
  - remember design B
10 Normalization

- Normalization
- Functional dependencies
- Normal forms
  - 1NF, 2NF, 3NF, BCNF
  - Higher normal forms
- Denormalization

\[ \pi_{\alpha_1}(\text{Hero}) \]

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Thor</td>
</tr>
<tr>
<td>2</td>
<td>Mister Fantastic</td>
</tr>
<tr>
<td>3</td>
<td>Iron Man</td>
</tr>
<tr>
<td>4</td>
<td>Hulk</td>
</tr>
<tr>
<td>5</td>
<td>Captain America</td>
</tr>
<tr>
<td>6</td>
<td>Invisible Girl</td>
</tr>
</tbody>
</table>

\[ \pi_{\alpha_2}(\text{Hero}) \]

<table>
<thead>
<tr>
<th>team_id</th>
<th>team_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Avengers</td>
</tr>
<tr>
<td>2</td>
<td>Fantastic Four</td>
</tr>
</tbody>
</table>

\[ \pi_{\alpha_3}(\text{Hero}) \]

<table>
<thead>
<tr>
<th>hero_id</th>
<th>team_id</th>
<th>join_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1961</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1964</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1961</td>
</tr>
</tbody>
</table>
10.1 Normalization

• The *rules of thumb for good database design* can be formalized by the concept of relational database normalization

• But before going into details, let’s recap some definitions from the relational model

  – data is represented using a *relation schema* \( S(R_1, ..., R_n) \)
    • each relation \( R(A_1, ..., A_n) \) contains attributes \( A_1, ..., A_n \)

  – a *relational database schema* consists of
    • a set of relations
    • a set of *integrity constraints* (e.g. \( \text{hero_id} \) is unique and \( \text{hero_id} \) determines \( \text{hero_name} \))

  – a *relational database instance* (or extension) is
    • a set of tuples adhering to the respective schemas and respecting all integrity constraints
10.1 Normalization

• For this lecture, let’s assume the following
  – $S(R_1, ..., R_n)$ is a relation schema
  – $R(A_1, ..., A_n)$ is a relation in $S$
  – $\mathcal{C}$ is a set of constraints satisfied by all extensions of $S$

• Our ultimate goal is to enhance the database design by decomposing the relations in $S$ into a set of smaller relations, as we did in our example:
• **Definition (decomposition)**
  
  – let $\alpha_1, \ldots, \alpha_k \subseteq \{A_1, \ldots, A_n\}$ be $k$ subsets of $R$’s attributes
    
    • note that these subsets may be overlapping
  
  – then, for any $\alpha_i$, a new relation $R_i$ can be derived:
    
    $$R_i = \pi_{\alpha_i}(R)$$
    
    – $\alpha_1, \ldots, \alpha_k$ is called a **decomposition** of $R$
  
  • *Good* decompositions have to be **reversible**
    
    – the decomposition $\alpha_1, \ldots, \alpha_k$ is called **lossless** if and only if $R = R_1 \bowtie R_2 \bowtie \cdots \bowtie R_k$, for any extension of $R$ satisfying the constraints $\mathcal{C}$
10.1 Normalization

- Example

\(C = \{\text{\{hero\_id, team\_id\} is unique},\) 
\(\text{hero\_id determines hero\_name},\) 
\(\text{team\_id determines team\_name},\) 
\(\text{\{hero\_id, team\_id\} determines join\_year}\}\}

- our example decomposition is lossless

\(\alpha_1 = \{\text{heroID, heroname}\}, \quad \alpha_2 = \{\text{teamID, teamName}\}, \quad \alpha_3 = \{\text{heroID, teamID, joinYear}\}\)

\[\pi_{\alpha_1}(\text{Hero})\]

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Thor</td>
</tr>
<tr>
<td>2</td>
<td>Mister Fantastic</td>
</tr>
<tr>
<td>3</td>
<td>Iron Man</td>
</tr>
<tr>
<td>4</td>
<td>Hulk</td>
</tr>
<tr>
<td>5</td>
<td>Captain America</td>
</tr>
<tr>
<td>6</td>
<td>Invisible Girl</td>
</tr>
</tbody>
</table>

\[\pi_{\alpha_2}(\text{Hero})\]

<table>
<thead>
<tr>
<th>team_id</th>
<th>team_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>The Avengers</td>
</tr>
<tr>
<td>2</td>
<td>Fantastic Four</td>
</tr>
</tbody>
</table>

\[\pi_{\alpha_3}(\text{Hero})\]

<table>
<thead>
<tr>
<th>hero_id</th>
<th>team_id</th>
<th>join_year</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1961</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1963</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1964</td>
</tr>
<tr>
<td>6</td>
<td>2</td>
<td>1961</td>
</tr>
</tbody>
</table>
10.1 Normalization

\[ C = \{ \text{"\{hero_id, team_id\} is unique"}, \]
\[ \text{\"hero_id determines hero_name"}, \]
\[ \text{\"team_id determines team_name"}, \]
\[ \text{\"\{hero_id, team_id\} determines join_year"}\} \]

—is the following a lossless decomposition?

\[ \alpha_1 = \{ \text{hero_id, team_id, join_year}\}, \quad \alpha_2 = \{ \text{team_id, hero_name, team_name, join_year}\} \]

\[ \pi_{\alpha_1}(\text{Hero}) \]

\[ \begin{array}{ccc}
\text{hero_id} & \text{team_id} & \text{join_year} \\
1 & 1 & 1963 \\
2 & 2 & 1961 \\
3 & 1 & 1963 \\
4 & 1 & 1963 \\
5 & 1 & 1964 \\
6 & 2 & 1961 \\
\end{array} \]

\[ \pi_{\alpha_2}(\text{Hero}) \]

\[ \begin{array}{cccc}
\text{team_id} & \text{hero_name} & \text{team_name} & \text{join_year} \\
1 & \text{Thor} & \text{The Avengers} & 1963 \\
2 & \text{Mister Fantastic} & \text{Fantastic Four} & 1961 \\
3 & \text{Iron Man} & \text{The Avengers} & 1963 \\
4 & \text{Hulk} & \text{The Avengers} & 1963 \\
5 & \text{Captain America} & \text{The Avengers} & 1964 \\
6 & \text{Invisible Girl} & \text{Fantastic Four} & 1961 \\
\end{array} \]
10.1 Normalization

• **Normalizing** a relation schema $S$ means replacing relations in $S$ by lossless decompositions.

• However, this raises some new questions:
  – under which conditions is there a (nontrivial) lossless decomposition?
  • decompositions involving $\alpha_i = \{A_1, \ldots, A_n\}$ or $\alpha_i = \emptyset$ are called **trivial**
  – if there is a lossless decomposition, how to find it?
  – how to measure a relation schema’s design quality?
  • We may abstain from further normalization if the quality is good enough...
• The normalization of $S$ depends entirely on the set of constraints $\mathcal{C}$ imposed on $S$
• Instead of dealing with constraints of arbitrary complexity, we restrict $\mathcal{C}$ to the class of functional dependencies (FDs)
  – hero_name is completely determined by hero_id
    is an example for a functional dependency
  – most update anomalies and problems with redundancy occurring in practice can be traced back to violations of functional dependency constraints
    • typically, functional dependencies are all you need
Normalization

Functional dependencies

Normal forms
  - 1NF, 2NF, 3NF, BCNF
  - Higher normal forms

Denormalization
10.2 Functional Dependencies

• Informally, functional dependencies can be described as follows
  – let \( X \) and \( Y \) be some sets of attributes
  – if \( Y \) functionally depends on \( X \), and two tuples agree on their \( X \) values, then they also have to agree on their \( Y \) values

• Examples
  – \{end\_time\} functionally depends on \{start\_time, duration\}
  – \{duration\} functionally depends on \{start\_time, end\_time\}
  – \{end\_time\} functionally depends on \{end\_time\}
10.2 Functional Dependencies

Formal definition

• Let $X$ and $Y$ be subsets of $R$’s attributes
  – That is, $X, Y \subseteq \{A_1, ..., A_n\}$

• There is **functional dependency (FD)** between $X$ and $Y$ (denoted as $X \rightarrow Y$), if and only if, ...
  – … for any two tuples $t_1$ and $t_2$ within any instance of $R$, the following is true:

$$\text{If } \pi_X t_1 = \pi_X t_2, \text{ then } \pi_Y t_1 = \pi_Y t_2$$
10.2 Functional Dependencies

• If $X \rightarrow Y$, then one says that ...
  – $X$ functionally determines $Y$, and
  – $Y$ functionally depends on $X$.
• $X$ is called the determinant of the FD $X \rightarrow Y$
• $Y$ is called the dependent of the FD $X \rightarrow Y$
10.2 Functional Dependencies

• Functional dependencies are semantic properties of the underlying domain and data model
  – They depend on real world knowledge

• FDs are NOT a property of a particular instance (extension) of the relation schema!
  – the designer is responsible for identifying FDs
  – FDs are manually defined integrity constraints on S
  – all extensions respecting S’s functional dependencies are called legal extensions of S
10.2 Functional Dependencies

• In fact, functional dependencies are a generalization of key constraints

• To show this, we need a short recap
  – a set of attributes $X$ is a (candidate) key for $R$ if and only if it has both of the following properties
    • uniqueness: no legal instance of $R$ ever contains two distinct tuples with the same value for $X$
    • irreducibility: no proper subset of $X$ has the uniqueness property
  – a superkey is a superset of a key
    • i.e. only uniqueness is required
10.2 Functional Dependencies

• In practice, if there is more than one key, we usually choose one and call it the primary key – however, for normalization purposes, only keys are important – thus, we ignore primary keys today

• The following is true
  – $X$ is a superkey of $R$ if and only if $X \rightarrow \{A_1, ..., A_n\}$ is a functional dependency in $R$
• Example

– a relation containing students

• semantics: matriculation_no is unique
• \{matriculation_no\} → \{firstname, lastname, birthdate\}
10.2 Functional Dependencies

• Example

  – a relation containing real names and aliases of heroes, where each hero has only one unique alias

    • \{alias\} → \{real_name\}
10.2 Functional Dependencies

• Example

– a relation containing license plates and the type of the respective car

  • \{area\_code, character\_code, number\_code\} \rightarrow \{car\_type\}
Quick Summary on keys:

- **Candidate Key** (or simply key)
  - A *irreducible* set of attributes which *uniquely* identifies a tuple
    - i.e.: all non-key attributes are functional dependent on the key, and the key is minimal
- **Superkey** is a superset of a key
  - i.e. only uniqueness is required
    - Superkey also identifies all other attributes, but is not minimal
- **Primary Key**
  - A primary key is one single key chosen from the set of candidate keys by the database designer
    - This choice impacts the way the DBMS manages relations and queries
What FDs can be derived from the following description of an address book?

- for any given zip code, there is just one city and state
  - …which, to be exact, is not true in reality
- for any given street, city, and state, there is just one zip code.

FDs and candidate keys?
10.2 Functional Dependencies

• One possible solution:
  – \{zip\} → \{city, state\}
  – \{street, city, state\} → \{zip\}
• Typically, not all actual FDs are modeled explicitly
  – \{zip\} → \{city\}
  – \{street\} → \{street\}
  – \{state\} → ∅
  – ...

street  city  state  zip
10.2 Functional Dependencies

• Obviously, some FDs are implied by others
  – \{zip\} \rightarrow \{city, state\} implies \{zip\} \rightarrow \{city\}

• Moreover, some FDs are trivial
  – \{street\} \rightarrow \{street\}
  – \{state\} \rightarrow \emptyset
  – **definition**: The FD \( X \rightarrow Y \) is called trivial iff \( X \supseteq Y \)

• What do we need?
  – a **compact representation** for sets of FD constraints
    • no redundant FDs
  – an **algorithm** to compute the set of all implied FDs
• **Definition:**
For any set $F$ of FDs, the **closure** of $F$ (denoted $F^+$) is the set of all FDs that are logically **implied** by $F$

– *Abstract Definition*: $F$ **implies** the FD $X \rightarrow Y$, if and only if any extension of $R$ satisfying any FD in $F$, also satisfies the FD $X \rightarrow Y$

• Fortunately, the closure of $F$ can easily be computed using a small set of **inference rules**
For any attribute sets $X$, $Y$, $Z$, the following is true:

- **reflexivity:**
  \[ X \supseteq Y \implies X \rightarrow Y \]

- **augmentation:**
  \[ X \rightarrow Y \implies X \cup Z \rightarrow Y \cup Z \]

- **transitivity:**
  \[ X \rightarrow Y \text{ and } Y \rightarrow Z \implies X \rightarrow Z \]

These rules are called **Armstrong's axioms**

- one can show that they are **complete** and **sound**
  - **completeness:** every implied FD can be derived
  - **soundness:** no non-implied FD can be derived
To simplify the practical task of computing $F^+$ from $F$, several additional rules can be derived from Armstrong’s axioms:

– **decomposition:**
  If $X \rightarrow Y \cup Z$, then $X \rightarrow Y$ and $X \rightarrow Z$

– **union:**
  If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow Y \cup Z$

– **composition:**
  If $X \rightarrow Y$ and $Z \rightarrow W$, then $X \cup Z \rightarrow Y \cup W$
10.2 Functional Dependencies

• Example

– relational schema $R(A, B, C, D, E, F)$
– FDs: $\{A\} \rightarrow \{B, C\}$  $\{B\} \rightarrow \{E\}$  $\{C, D\} \rightarrow \{E, F\}$
– then we can make the following derivation

1. $\{A\} \rightarrow \{B, C\}$ (given)
2. $\{A\} \rightarrow \{C\}$ (by decomposition)
3. $\{A, D\} \rightarrow \{C, D\}$ (by augmentation)
4. $\{A, D\} \rightarrow \{E, F\}$ (by transitivity with given $\{C, D\} \rightarrow \{E, F\}$)
5. $\{A, D\} \rightarrow \{F\}$ (by decomposition)
• In principle, we can compute the closure $F^+$ of a given set $F$ of FDs by means of the following algorithm:
  – *Repeatedly apply the six inference rules until they stop producing new FDs.*

• In practice, this algorithm is hardly very efficient
  – however, there usually is little need to compute the full closure
  – instead, it often suffices to compute a certain subset of the closure: the subset consisting of all FDs with given left side
    • This will later serve for finding proper keys or normalizing relations
10.2 Functional Dependencies

- **Definition:**
  Given a set of attributes $X$ and a set of FDs $F$, the **closure** of $X$ under $F$, written as $(X, F)^+$, consists of all attributes that functionally depend on $X$
  - i.e. $(X, F)^+ := \{A_i \mid X \rightarrow A_i \text{ is implied by } F\}$

- The following algorithm computes $(X, F)^+$:

```plaintext
unused := F
_closure := X
do {
    for $(Y \rightarrow Z) \in \text{unused}$ do {
        if $(Y \subseteq \text{closure})$ do {
            unused := unused \ {$(Y \rightarrow Z)$}
            closure := closure U Z
        }
    }
} while (unused and closure did not change)
return closure
```
• Quiz

$F = \{ \{A\} \rightarrow \{B, C\}, \{B\} \rightarrow \{E\}, \{E\} \rightarrow \{C, F\}, \{C, D\} \rightarrow \{E, F\}\}$

What is the closure of $\{A, B\}$ under $F$?

```plaintext
unused := F
closure := X
do {
  for (Y → Z ∈ unused) {
    if (Y ⊆ closure) {
      unused := unused \ {Y → Z}
      closure := closure U Z
    }
  }
} while (unused and closure did not change)
return closure
```
• Quiz

\[ F = \{ \{A\} \rightarrow \{B, C\}, \{B\} \rightarrow \{E\}, \{E\} \rightarrow \{C, F\}, \{C, D\} \rightarrow \{E, F\} \} \]

What is the closure of \(\{A, B\}\) under \(F\)?

\[
\left(\{A, B\}, F\right)^+ = \{A, B, C, E, F\}
\]

Add C, because \(\{A\} \rightarrow \{C\}\)

Add E, because \(\{B\} \rightarrow \{E\}\)

Add F, because \(\{E\} \rightarrow \{F\}\)
10.2 Functional Dependencies

• Now, we can do the following
  – given a set $F$ of FDs, we can easily tell whether a specific FD $X \rightarrow Y$ is contained in $F^+$
    • just check whether $Y \subseteq (X, F)^+$
  – in particular, we can find out whether a set of attributes $X$ is a superkey of $R$
    • just check whether $(X, F)^+ = \{A_1, ..., A_n\}$

• What’s still missing?
  – given a set of FDs $F$, how to find a set of FDs $G$, such that $F^+ = G^+$, and $G$ is as small as possible?
  – given sets of FDs $F$ and $G$, does $F^+ = G^+$ hold?
10.2 Functional Dependencies

- **Definition:**
  Two sets of FDs $F$ and $G$ are equivalent iff $F^+ = G^+$

- How can we find out whether two given sets of FDs $F$ and $G$ are equivalent?
  - **Theorem:**
    $F^+ = G^+$ iff for any FD $X \rightarrow Y \in F \cup G$, it is $(X, F)^+ = (X, G)^+$
  - **Proof**
    - let $F' = \{X \rightarrow (X, F)^+ \mid X \rightarrow Y \in F \cup G\}$
    - analogously, derive $G'$ from $G$
    - obviously, then $F'^+ = F^+$ and $G'^+ = G^+$
    - moreover, every left side of an FD in $F'$ occurs as a left side of an FD in $G'$ (and reverse)
    - if $F'$ and $G'$ are different, then also $F^+$ and $G^+$ must be different
### 10.2 Functional Dependencies

**Example**

- \( F = \{ \{A, B\} \rightarrow \{C\}, \{C\} \rightarrow \{B\} \} \)
- \( G = \{ \{A\} \rightarrow \{C\}, \{A, C\} \rightarrow \{B\} \} \)
- are \( F \) and \( G \) equivalent?

- we must check \((X, F)^+ = (X, G)^+\) for the following \( X \)
  - \( \{A, B\}, \{C\}, \{A\}, \text{ and } \{A, C\} \)

- \((\{A, B\}, F)^+ = \{A, B, C\}\) \((\{A, B\}, G)^+ = \{A, B, C\}\)
- \((\{C\}, F)^+ = \{B, C\}\) \((\{C\}, G)^+ = \{C\}\)

- therefore, \( F \) and \( G \) are not equivalent!
10.2 Functional Dependencies

• **Remember:**
  To have a **small representation** of $F$, we want to find a $G$, such that
  - $F$ and $G$ are equivalent
  - $G$ is as *small as possible* (we will call this property **minimality**)

• **Definition:**
  A set of FDs $F$ is **minimal** iff the following is true
  - every FD $X \rightarrow Y$ in $F$ is in **canonical form**
    - i.e. $Y$ consists of exactly one attribute
  - every FD $X \rightarrow Y$ in $F$ is **left-irreducible**
    - i.e. no attribute can be removed from $X$ without changing $F^+$
  - every FD $X \rightarrow Y$ in $F$ is **non-redundant**
    - i.e. $X \rightarrow Y$ cannot be removed from $F$ without changing $F^+$
The following algorithm *minimizes* $F$, that is, it transforms $F$ into a minimal equivalent of $F$:

1. Split up all right sides to get FDs in canonical form.
2. Remove all redundant attributes from the left sides (by checking which attribute removals change $F^+$).
3. Remove all redundant FDs from $F$ (by checking which FD removals change $F^+$).
10.2 Functional Dependencies

• Example

- given \( F = \{ \{A\} \rightarrow \{B, C\}, \{B\} \rightarrow \{C\},\{A\} \rightarrow \{B\}, \{A, B\} \rightarrow \{C\}, \{A, C\} \rightarrow \{D\} \} \)

1. Split up the right sides:

\( \{A\} \rightarrow \{B\}, \{A\} \rightarrow \{C\}, \{B\} \rightarrow \{C\}, \{A, B\} \rightarrow \{C\}, \{A, C\} \rightarrow \{D\} \)

2. Remove \( C \) from \( \{A, C\} \rightarrow \{D\} \):

- \( \{A\} \rightarrow \{C\} \) implies \( \{A\} \rightarrow \{A, C\} \) (augmentation)
- \( \{A\} \rightarrow \{A, C\} \) and \( \{A, C\} \rightarrow \{D\} \) imply \( \{A\} \rightarrow \{D\} \) (transitivity)
Now we have:

\{A\} \rightarrow \{B\}, \quad \{A\} \rightarrow \{C\}, \quad \{B\} \rightarrow \{C\},
\{A, B\} \rightarrow \{C\}, \quad \{A\} \rightarrow \{D\}

3. Remove \{A, B\} \rightarrow \{C\}:
   - \{A\} \rightarrow \{C\} implies \{A, B\} \rightarrow \{C\}

4. Remove \{A\} \rightarrow \{C\}:
   - \{A\} \rightarrow \{B\} and \{B\} \rightarrow \{C\} imply \{A\} \rightarrow \{C\} (transitivity)

Finally, we end up with a **minimal equivalent** of \(F\):

\{A\} \rightarrow \{B\}, \quad \{B\} \rightarrow \{C\}, \quad \{A\} \rightarrow \{D\}
10.2 Functional Dependencies

• Functional dependencies are the perfect tool for performing lossless decompositions

  – Heath’s Theorem:

    Let \( X \rightarrow Y \) be an FD constraint of the relation schema \( R(A_1, ..., A_n) \). Then, the following decomposition of \( R \) is lossless:

    \[
    \alpha_1 = X \cup Y \quad \text{and} \quad \alpha_2 = \{A_1, ..., A_n\} \setminus Y.
    \]

  – Example:

    | hero_id | team_id | hero_name | team_name | join_year |
    |---------|---------|-----------|-----------|-----------|
    |         |         |           |           |           |
    |         |         |           |           |           |

    FDs:
    - \( \{\text{hero_id}\} \rightarrow \{\text{hero_name}\} \)
    - \( \{\text{team_id}\} \rightarrow \{\text{team_name}\} \)
    - \( \{\text{hero_id, team_id}\} \rightarrow \{\text{join_year}\} \)

    Decompose with respect to
    - \( \{\text{hero_id}\} \rightarrow \{\text{hero_name}\} \)
How to come up with functional dependencies?

- there are several ways
  - Based on domain knowledge
  - Based on an explicit data model
  - Based on existing data

1. Based on domain knowledge
   - obvious FDs are easy to find
   - what about more complicated FDs?
   - no guarantee that you found all (important) FDs!
2. Based on an explicit model

- automated generation of FDs possible
- but: are all actual FDs present in the model?
  - what about FDs between attributes of the same entity?
3. Based on existing data
   – in practice, often there is already some data available (that is, tuples)
   – we can use the data to derive FD constraints
   – obviously
     • all FDs that hold in general for some relation schema, also hold for any given extension
     • therefore, the set of all FDs that hold in some extension, is a superset of all true FDs of the relation schema
   – what we can do
     • find all FDs that hold in a given extension
     • find a minimal representation of this FD set
     • ask a domain expert, what FDs are generally true
### 10.2 Functional Dependencies

Which of the following FDs are satisfied in this particular extension?

- a) \( \{C\} \rightarrow \{A, B\} \)
- b) \( \{A, D\} \rightarrow \{C\} \)
- c) \( \{\} \rightarrow \{E\} \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
10.2 Functional Dependencies

find all FDs that are satisfied in this extension!

- we will check any FD $X \rightarrow Y$ in canonical form, i.e.,
  $X$ is a subset of $\{A, B, C, D, E\}$ and
  $Y$ is an element of $\{A, B, C, D, E\}$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
10 Normalization

- Normalization
- Functional dependencies
- **Normal forms**
  - 1NF, 2NF, 3NF, BCNF
  - Higher normal forms
- Denormalization
10.3 Normal Forms

• Back to normalization
  – remember:
    normalization = finding lossless decompositions
  – but only decompose, if the relation schema is of bad quality

• How to measure the quality of a relation schema?
  – clearly: the quality depends on the constraints
  – in our case:
    quality depends on the FDs of the relation schema
  – schemas can be classified into different quality levels, which are called normal forms
Part of a schema design process is to choose a desired normal form and convert the schema into that form.

There are seven normal forms:

- the higher the number, ...
  - ... the stricter the requirements,
  - ... the less anomalies and redundancy, and
  - ... the better the design quality.
    - (well, from a theoretical point of view; in the real world, there might be good reasons why a lower normal form is better.)
10.3 1NF

• First normal form (1NF)
  – already known from previous lectures
    • has nothing to do with functional dependencies!
  – restricts relations to being flat
    • only atomic attribute values are allowed
  – multi-valued attributes must be normalized, e.g., by
    a) introducing a new relation for the multi-valued attribute
    b) replicating the tuple for each multi-value
    c) introducing an own attribute for each multi-value
      (if there is a small maximum number of values)
  – solution a) is usually considered the best
a) Introducing a **new relation**

- uses old key and multi-attribute as composite key

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
<th>powers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Storm</td>
<td>weather control, flight</td>
</tr>
<tr>
<td>2</td>
<td>Wolverine</td>
<td>extreme cellular regeneration</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>omnipotence, indestructibility, limitless energy manipulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
<th>power</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Storm</td>
<td>weather control</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>flight</td>
</tr>
<tr>
<td>2</td>
<td>Wolverine</td>
<td>extreme cellular regeneration</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>omnipotence</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>indestructibility</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>limitless energy manipulation</td>
</tr>
</tbody>
</table>
b) **Replicating** the tuple for each multi-value

– uses old key and multi-attribute as composite key

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
<th>powers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Storm</td>
<td>weather control, flight</td>
</tr>
<tr>
<td>2</td>
<td>Wolverine</td>
<td>extreme cellular regeneration</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>omnipotence, indestructibility, limitless energy manipulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
<th>powers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Storm</td>
<td>weather control</td>
</tr>
<tr>
<td>1</td>
<td>Storm</td>
<td>flight</td>
</tr>
<tr>
<td>2</td>
<td>Wolverine</td>
<td>extreme cellular regeneration</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>omnipotence</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>indestructibility</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>limitless energy manipulation</td>
</tr>
</tbody>
</table>
c) Introducing an **own attribute** for each multi-value

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
<th>powers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Storm</td>
<td>weather control, flight</td>
</tr>
<tr>
<td>2</td>
<td>Wolverine</td>
<td>extreme cellular regeneration</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>omnipotence, indestructibility, limitless energy manipulation</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
<th>power1</th>
<th>power2</th>
<th>power3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Storm</td>
<td>weather control</td>
<td>flight</td>
<td>NULL</td>
</tr>
<tr>
<td>2</td>
<td>Wolverine</td>
<td>cellular regeneration</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>3</td>
<td>Phoenix</td>
<td>omnipotence</td>
<td>indestructibility</td>
<td>limitless energy manipulation</td>
</tr>
</tbody>
</table>
10.3 2NF

• The second normal form (2NF)
  – the 2NF aims to avoid attributes that are functionally dependent on (proper) subsets of keys
  – remember
    • a set of attributes $X$ is a \textit{(candidate) key} if and only if $X \rightarrow \{A_1, ..., A_n\}$ is a valid FD
    • an attribute $A_i$ is a \textit{key attribute} if and only if it is contained in some key; otherwise, it is a \textit{non-key attribute}
  – definition (2NF):
    A relation schema is in \textbf{2NF} (wrt. a set of FDs) iff ...
    • it is in 1NF and
    • no non-key attribute is functionally dependent on a proper subset of some key.
• Functional dependence on key parts is only a problem in relation schemas with composite keys
  – a key is called **composite key** if it consists of more than one attribute

• **Corollary:**
  Every 1NF-relation without constant attributes and without **composite keys** also is in 2NF.
  – 2NF is violated, if there is a **composite key** and some **non-key attribute** depends only on a **proper subset** of this composite key
### 10.3 2NF

**Normalization** into 2NF is archived by **decomposition** according to the *non-2NF* FDs

- if \( X \rightarrow Y \) is a valid FD and \( X \) is a proper subset of some key, then decompose into \( \alpha_1 = X \cup Y \) and \( \alpha_2 = \{A_1, ..., A_n\} \setminus Y \)
- according to Heath’s Theorem, this decomposition is **lossless**

**FDs:**

- \( \{\text{hero_id}\} \rightarrow \{\text{hero_name}\} \)
- \( \{\text{team_id}\} \rightarrow \{\text{team_name}\} \)
- \( \{\text{hero_id, team_id}\} \rightarrow \{\text{join_year}\} \)

**Decompose with respect to**

- \( \{\text{hero_id}\} \rightarrow \{\text{hero_name}\} \)
10.3 2NF

- Repeat this decomposition step for every created relation schema that is still not in 2NF

**FDs:**

\{team_id\} → \{team_name\}
\{hero_id, team_id\} → \{join_year\}

Decompose with respect to
\{team_id\} → \{team_name\}
The third normal form (3NF)

- the 3NF relies on the concept of **transitive FDs**

- **definition:**
  Given a set of FDs $F$, an FD $X \rightarrow Z \in F^+$ is **transitive** in $F$, if and only if there is an attribute set $Y$ such that:
  
  $X \rightarrow Y \in F^+$,
  $Y \rightarrow X \notin F^+$, and
  $Y \rightarrow Z \in F^+$.

- **Example**
  
  - $\{\text{hero_id}\} \rightarrow \{\text{hero_name}\}$
  - $\{\text{hero_id}\} \rightarrow \{\text{home_city_id}\}$
  - $\{\text{hero_id}\} \rightarrow \{\text{home_city_name}\}$
  - $\{\text{home_city_id}\} \rightarrow \{\text{home_city_name}\}$
• **Definition:**
A relation schema is in 3NF if and only if:

– it is 2NF and

– no key transitively determines a non-key attribute.

<table>
<thead>
<tr>
<th>hero_id</th>
<th>hero_name</th>
<th>home_city_id</th>
<th>home_city_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Professor X</td>
<td>563</td>
<td>New York</td>
</tr>
<tr>
<td>12</td>
<td>Wolverine</td>
<td>782</td>
<td>Alberta</td>
</tr>
<tr>
<td>13</td>
<td>Cyclops</td>
<td>112</td>
<td>Anchorage</td>
</tr>
<tr>
<td>14</td>
<td>Phoenix</td>
<td>563</td>
<td>New York</td>
</tr>
</tbody>
</table>

{hero_id} → {hero_name}
{hero_id} → {home_city_id}
{home_city_id} → {home_city_name}
10.3 3NF

- Assume that the non-3NF transitive FD $X \rightarrow Z$ has been created by FDs $X \rightarrow Y$ and $Y \rightarrow Z$
- Then, normalization into 3NF is archived by decomposition according to $Y \rightarrow Z$
  - again, this decomposition is lossless

FDs:
- $\{\text{hero}._\text{id}\} \rightarrow \{\text{hero}._\text{name}\}$
- $\{\text{hero}._\text{id}\} \rightarrow \{\text{home}._\text{city}._\text{id}\}$
- $\{\text{home}._\text{city}._\text{id}\} \rightarrow \{\text{home}._\text{city}._\text{name}\}$

Decompose with respect to $\{\text{home}._\text{city}._\text{id}\} \rightarrow \{\text{home}._\text{city}._\text{name}\}$
10.3 BCNF

• Boyce-Codd normal form (BCNF)
  – was actually proposed by Ian Heath (he called it 3NF) three years before Boyce and Codd
  – definition:
    A relation schema $R$ is in **BCNF** if and only if, in any non-trivial FD $X \rightarrow Y$, the set $X$ is a superkey

• All BCNF schemas are also in 3NF, and most 3NF schemas are also in BCNF
  – there are some rare exceptions
10.3 BCNF

– BCNF is very similar to 3NF:

  • **BCNF:**
    In any non-trivial FD $X \rightarrow Y$, the set $X$ is a superkey.
  
  • **3NF (alternative definition):**
    In any non-trivial FD $X \rightarrow Y$, the set $X$ is a superkey, or the set $Y$ is a subset of some key.

– a 3NF schema is not in BCNF, if it has two or more overlapping composite keys.

  • i.e. there are different keys $X$ and $Y$ such that $|X|, |Y| \geq 2$ and $X \cap Y \neq \emptyset$. 
• **Example**

  – *Students, a topic, and an advisor*
  
  – let’s assume that the following dependencies hold
    
    - \{student, topic\} → \{advisor\}
    - \{advisor\} → \{topic\}
  
  – i.e. *For each topic, a student has a specific advisor. Each advisor is responsible for a single specific topic.*
– consequently, there are the following keys
  • \{\text{student, topic}\}
  • \{\text{student, advisor}\}

– the schema is in 3\text{NF}, because it is in 1\text{NF} and there are no non-key attributes

– however, it is not in BC\text{NF}
  • We have \{\text{advisor}\} \rightarrow \{\text{topic}\} but \{\text{advisor}\} is not a superkey
Moreover, there are **modification anomalies**:

<table>
<thead>
<tr>
<th>student</th>
<th>topic</th>
<th>advisor</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Math</td>
<td>Gauss</td>
</tr>
<tr>
<td>100</td>
<td>Physics</td>
<td>Einstein</td>
</tr>
<tr>
<td>101</td>
<td>Math</td>
<td>Leibniz</td>
</tr>
<tr>
<td>102</td>
<td>Math</td>
<td>Gauss</td>
</tr>
</tbody>
</table>

If you delete this row, all information about Leibniz doing math is lost.

What options do we have?

- decompose into one of
  - **student** | topic and **student** | advisor
  - topic | **advisor** and topic | **student**
  - **advisor** | topic and **advisor** | **student**

- Which one to choose?

- \{Student, Topic\} → \{Advisor\} is "lost" in all options
In any case, we should perform a lossless decomposition

- Apply Heath’s theorem w.r.t. \{advisor\} $\rightarrow$ \{topic\}
  - $\Rightarrow$ advisor topic and advisor student

- All other decompositions can produce false tuples when rejoining

- Completeness of FDs was traded against a higher normal form
• BCNF is the *ultimate* normal form when using only functional dependencies as constraints
  – “*Every attribute depends on a key, a whole key, and nothing but a key, so help me Codd.*”

• However, there are higher normal forms (4NF to 6NF) that rely on generalizations of FDs
  – 4NF: multivalued dependencies
  – 5NF/6NF: join dependencies
The 4NF is about multivalued dependencies (MVDs)

Example

<table>
<thead>
<tr>
<th>course</th>
<th>teacher</th>
<th>textbook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physics</td>
<td>Prof. Green</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Physics</td>
<td>Prof. Green</td>
<td>Principles of Optics</td>
</tr>
<tr>
<td>Physics</td>
<td>Prof. Brown</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Physics</td>
<td>Prof. Brown</td>
<td>Principles of Optics</td>
</tr>
<tr>
<td>Math</td>
<td>Prof. Green</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Math</td>
<td>Prof. Green</td>
<td>Vector Analysis</td>
</tr>
<tr>
<td>Math</td>
<td>Prof. Green</td>
<td>Trigonometry</td>
</tr>
</tbody>
</table>

Dependencies:

- For any course, there is a fixed set of teachers.
  (written as \{course\} \rightarrow \{teacher\})
- For any course, there is a fixed set of textbooks, which is independent of the teacher.
  (written as \{course\} \rightarrow \{textbook\})

In fact, every FD can be expressed as a MVD
- if \( X \rightarrow Y \) then also \( X \rightarrow\{ Y \} \)
- but both expressions are not equivalent!
• **Definition:**
  A relation schema is in 4NF if and only if, for any non-trivial multivalued dependency \(X \rightarrow Y\), also the functional dependency \(Z \rightarrow Y\) holds, for some key \(Z\)

• **Decomposition into 4NF schemas:**

<table>
<thead>
<tr>
<th>course</th>
<th>teacher</th>
<th>textbook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physics</td>
<td>Prof. Green</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Physics</td>
<td>Prof. Green</td>
<td>Principles of Optics</td>
</tr>
<tr>
<td>Physics</td>
<td>Prof. Brown</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Physics</td>
<td>Prof. Brown</td>
<td>Principles of Optics</td>
</tr>
<tr>
<td>Math</td>
<td>Prof. Green</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Math</td>
<td>Prof. Green</td>
<td>Vector Analysis</td>
</tr>
<tr>
<td>Math</td>
<td>Prof. Green</td>
<td>Trigonometry</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course</th>
<th>textbook</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physics</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Physics</td>
<td>Principles of Optics</td>
</tr>
<tr>
<td>Math</td>
<td>Basic Mechanics</td>
</tr>
<tr>
<td>Math</td>
<td>Vector Analysis</td>
</tr>
<tr>
<td>Math</td>
<td>Trigonometry</td>
</tr>
</tbody>
</table>
• Result from a bad conceptual schema:

- Instead of
• The 5NF deals with join dependencies (JDs)
  – directly related to lossless decompositions
  – definition:
    Let $\alpha_1, ..., \alpha_k \subseteq \{A_1, ..., A_n\}$ be $k$ subsets of $R$’s attributes (possibly overlapping). We say that $R$ satisfies the join dependency $\star \{\alpha_1, ..., \alpha_k\}$ if and only if $\alpha_1, ..., \alpha_k$ is a lossless decomposition of $R$.
  – definition:
    A relation schema is in 5NF if and only if, for every non-trivial join dependency $\star \{\alpha_1, ..., \alpha_k\}$, each $\alpha_i$ is a superkey.
• The 6NF also is about join dependencies
  – definition:
    A relation schema is in 6NF if and only if it satisfies no non-trivial JDs at all.
    • in other words: You cannot decompose it anymore.

• Decomposition into 6NF means that every resulting relation schema contains a key and one(!) additional non-key attribute
  – this means a lot of tables!

• By definition, 6NF is the final word on normalization by lossless decomposition
  – all kinds of dependencies can be expressed by key and foreign key constraints
10 Normalization

• Normalization
• Functional dependencies
• Normal forms
  – 1NF, 2NF, 3NF, BCNF
  – Higher normal forms
• Denormalization
10.4 Denormalization

• Normalization in real world databases
  – guided by normal form theory
  – but: normalization is not everything!
  – trade-off: redundancy/anomalies vs. speed
    • general design: avoid redundancy wherever possible, because redundancies often lead to inconsistent states
    • an exception: materialized views (≈ precomputed joins) – expensive to maintain, but can boost read efficiency
    • Also: distributed and parallel databases
      – Here, redundancy is a good thing and increases data reliability and query speeds!
        » .but creates huge problems when faced with updates…
• Usually, a schema in a higher normal form is better than one in a lower normal form
  – however, sometimes it is a good idea to artificially create lower-form schemas to, e.g., increase read performance
    • this is called denormalization
    • denormalization sometimes increases query speed and decreases update efficiency due to the introduction of redundancy
10.4 Denormalization

• Rules of thumb
  – a **good data model** almost always directly leads to relational schemas in high normal forms
    • carefully design your models!
    • think of dependencies and other constraints!
    • have normal forms in mind during modeling!
  – denormalize only when faced with a performance problem that cannot be resolved by
    • money
    • hardware scalability
    • current SQL technology
    • network optimization
    • parallelization
    • other performance techniques
10.4 Denormalization

– sometimes, you even can perform denormalization at the physical level of the database

- let your RDBMS know what attributes are often accessed together, even if they are located in different tables
- state-of-the-art RDBMS can exploit this information to physically cluster data or precompute some joins, even without changing your table designs!
• Advanced **database concepts** for application programming
  – **Views**
  – **Indexes**
  – **Transactions**

• Accessing databases from applications
  – Embedded SQL
  – SQLJ