• Relational database design requires that we find a “good” collection of relation schemas
• Functional dependencies can be used to refine ER diagrams, or independently (i.e., by performing repetitive decompositions on a "universal" relation that contains all attributes)
• A bad design may lead to several problems
Assume the position determines the salary:

position $\rightarrow$ salary

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
<th>address</th>
<th>department</th>
<th>position</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dewi</td>
<td>Srijaya</td>
<td>12a Jln Lempeng</td>
<td>Toys</td>
<td>clerk</td>
<td>2000</td>
</tr>
<tr>
<td>Izabel</td>
<td>Leong</td>
<td>10 Outram Park</td>
<td>Sports</td>
<td>trainee</td>
<td>1200</td>
</tr>
<tr>
<td>John</td>
<td>Smith</td>
<td>107 Clementi Rd</td>
<td>Toys</td>
<td>clerk</td>
<td>2000</td>
</tr>
<tr>
<td>Axel</td>
<td>Bayer</td>
<td>55 Cuscaden Rd</td>
<td>Sports</td>
<td>trainee</td>
<td>1200</td>
</tr>
<tr>
<td>Winny</td>
<td>Lee</td>
<td>10 West Coast Rd</td>
<td>Sports</td>
<td>manager</td>
<td>2500</td>
</tr>
<tr>
<td>Sylvia</td>
<td>Tok</td>
<td>22 East Coast Lane</td>
<td>Toys</td>
<td>manager</td>
<td>2600</td>
</tr>
<tr>
<td>Eric</td>
<td>Wei</td>
<td>100 Jurong drive</td>
<td>Toys</td>
<td>assistant manager</td>
<td>2200</td>
</tr>
</tbody>
</table>

Redundant storage
Update anomaly
Potential deletion anomaly
Insertion anomaly
## Decomposition Example

### T2

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
<th>address</th>
<th>department</th>
<th>position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dewi</td>
<td>Srijaya</td>
<td>12a Jln lempeng</td>
<td>Toys</td>
<td>clerk</td>
</tr>
<tr>
<td>Izabel</td>
<td>Leong</td>
<td>10 Outram Park</td>
<td>Sports</td>
<td>trainee</td>
</tr>
<tr>
<td>John</td>
<td>Smith</td>
<td>107 Clementi Rd</td>
<td>Toys</td>
<td>clerk</td>
</tr>
<tr>
<td>Axel</td>
<td>Bayer</td>
<td>55 Cuscaden Rd</td>
<td>Sports</td>
<td>trainee</td>
</tr>
<tr>
<td>Winny</td>
<td>Lee</td>
<td>10 West Coast Rd</td>
<td>Sports</td>
<td>manager</td>
</tr>
<tr>
<td>Sylvia</td>
<td>Tok</td>
<td>22 East Coast Lane</td>
<td>Toys</td>
<td>manager</td>
</tr>
<tr>
<td>Eric</td>
<td>Wei</td>
<td>100 Jurong drive</td>
<td>Toys</td>
<td>assistant manager</td>
</tr>
</tbody>
</table>

### T3

<table>
<thead>
<tr>
<th>position</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>clerk</td>
<td>2000</td>
</tr>
<tr>
<td>trainee</td>
<td>1200</td>
</tr>
<tr>
<td>manager</td>
<td>2500</td>
</tr>
<tr>
<td>assistant manager</td>
<td>2200</td>
</tr>
<tr>
<td>security guard</td>
<td>1500</td>
</tr>
</tbody>
</table>

- No Redundant storage
- No Update anomaly
- No Deletion anomaly
- No Insertion anomaly
Normalization

- **Normalization** is the process of decomposing a relation schema $R$ into **fragments** (i.e., smaller tables) $R_1, R_2, \ldots, R_n$. Our goals are:
  
  - **Lossless decomposition**: The fragments should contain the same information as the original table. Otherwise decomposition results in information loss.

  - **Dependency preservation**: Dependencies should be preserved within each $R_i$, i.e., otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.

  - **Good form**: The fragments $R_i$ should not involve redundancy. Roughly speaking, a table has redundancy if there is a FD where the LHS is not a key (more on this later).
The decomposition is **lossless** (aka **lossless join**) if we can recover the initial table $T_1$ from fragments $T_2$, $T_3$:

```sql
select first_name, last_name, address, department, T2.position, salary
from T2, T3
where T2.position = T3.position
```

In general a decomposition of $R$ into $R_1$ and $R_2$ is **lossless** if and only if at least one of the following dependencies is in $F^+$:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

In other words, the common attribute of $R_1$ and $R_2$ must be a candidate key for $R_1$ or $R_2$.

In our example, the decomposition is lossless because `position` is a key for $T_3$. 

---

kwtleung@cse.ust.hk  
CSIT5300 (Spring 2017)
Example of Lossy Decomposition

- Decompose \( R = (A,B,C) \) into \( R_1 = (A,B) \) and \( R_2 = (B,C) \)

\[
\begin{array}{ccc}
A & B & C \\
\hline
a & 1 & m \\
a & 2 & n \\
b & 1 & p \\
\end{array}
\quad
\pi_{A,B}(R) \quad \pi_{B,C}(R)
\begin{array}{ccc}
A & B \\
\hline
a & 1 \\
a & 2 \\
b & 1 \\
\end{array}
\quad
\begin{array}{cc}
B & C \\
\hline
1 & m \\
2 & n \\
1 & p \\
\end{array}
\]

It is a lossy decomposition:
- Two extraneous tuples
- You get more, not less!!
- \( B \) is not a key of either fragment
• Suppose that the decomposition of a relation schema \( R \) with FDs \( F \) is a set of tables (fragments) \( R_i \), each having FDs \( F_i \)
  - \( F_i \) is the \textit{subset} of dependencies in \( F^+ \) (the closure of \( F \)) that include only attributes in \( R_i \).

The decomposition is \textit{dependency preserving} if and only if

\[(\bigcup_i F_i)^+ = F^+\]
R = (A, B, C), F = {{A}→{B}, {B}→{C}, {A}→{C}}. **Key:** A

There is a dependency {B}→{C}, where the LHS is not the key, meaning that there can be considerable redundancy in R.

**Solution:** Break it in two tables R1(A,B), R2(A,C) (normalization)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>

The decomposition is **lossless** because the common attribute A is a key for R1 (and R2)

The decomposition is **not dependency preserving** because $F_1={{A}→{B}}$, $F_2={{A}→{C}}$ and $(F_1∪F_2)^+≠F^+$. We lost the FD {B}→{C}.

In practical terms, each FD is implemented as an **assertion**, which is checked when there are updates. In the above example, in order to find violations, we have to join R1 and R2. This can be very expensive.
R = (A, B, C), F = {{A}→{B}, {B}→{C}, {A}→{C}}. Key: A

Break R in two tables R1(A,B), R2(B,C)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

The decomposition is **lossless** because the common attribute B is a key for R2.

The decomposition is **dependency preserving** because F1={{A}→{B}}, F2={{B}→{C}} and (F1∪F2)+=F+ ({A}→{C} can be found due to transitivity).

Violations can be found by inspecting the individual tables, without performing a join!
• Recall that the goal of a good database design are
  – **Lossless decomposition** - necessary in order to ensure correctness of the data
  – **Dependency preservation** – not necessary, but desirable in order to achieve efficiency of updates
  – **Good form** – desirable in order to avoid redundancy.

• But what it means for a table to be in good form?

• If the domains of all attributes in a table contain only atomic values, then the table is in **First Normal Form (1NF)**.
  – In other words, there are no nested tables, multi-valued attributes, or complex structures such as lists.

• Relational tables are always in **1NF**, according to the definition of the relational model.
A prime attribute is an attribute that is part of a candidate key.

Let $R$ be a relation schema, with the set $F$ of FDs. $R$ is in 2NF if and only if:
- for each FD: $X \rightarrow A$ in $F^+$

Then:
- $A \in X$ (the FD is trivial), or
- $X$ is not a proper subset of a candidate key for $R$, or
- $A$ is a prime attribute

**Note:** In 2NF, a subset of a candidate key cannot determine a non-prime attribute.

**Hint:** Whenever you try to determine the normal form (2NF, 3NF, BCNF) of a table, you **always** have to find all candidate keys.
Consider the relation scheme R=(A,B,C) with the FDs:
- \{A\} \rightarrow \{B\} and
- \{B\} \rightarrow \{C\}

- \{A\} is the only candidate key
- \{B\} is not a proper subset of the candidate key
- This scheme is in 2NF
Consider the relation schema $R=(A,B,C,D)$ with the FDs:
- $\{A,B\} \rightarrow \{C,D\}$ and
- $\{A\} \rightarrow \{D\}$

- $\{A,B\}$ is a candidate key (it is not a proper subset)
- $\{A\}$ is a proper subset of a candidate key
- $\{D\}$ is not a prime attribute
- This scheme is not in 2NF because of $\{A\} \rightarrow \{D\}$

**Note:** 2NF is not important because we can always achieve a better form (3NF – discussed next) that is lossless, preserves dependencies and contains less redundancy.
Let $R$ be a relation schema, with the set $F$ of FDs. $R$ is in **3NF** if and only if

- for each FD: $X \rightarrow A$ in $F^+$

Then
- $A \in X$ (the FD is trivial), or
- $X$ is a superkey for $R$, or
- $A$ is a prime attribute

**In words:** For every FD that does not contain extraneous (useless) attributes:
- the LHS is a candidate key, or
- the RHS is a prime attribute, i.e., it is an attribute that is part of a candidate key
3NF – Example

- \( R = (B, C, E) \)
  \[ F = \{ \{E\} \rightarrow \{B\}, \{B, C\} \rightarrow \{E\} \} \]

- Remember that you always have to find all candidate keys in order to determine the normal form of a table

- **Two candidate keys:** BC and EC
  - \( \{E\} \rightarrow \{B\} \) B is a prime attribute
  - \( \{B, C\} \rightarrow \{E\} \) BC is a candidate key

- None of the FDs violates the rules of the previous slide
  - R is in 3NF
• Let’s put the example of the previous slide into an application context

• Bank-schema = (Branch B, Customer C, Employee E)
  - \( \{E\} \rightarrow \{B\} \), e.g., an employee works in a single branch
  - \( \{B,C\} \rightarrow \{E\} \), e.g., when a customer goes to a certain branch s/he is always served by the same employee

<table>
<thead>
<tr>
<th>Branch</th>
<th>Customer</th>
<th>Employee</th>
</tr>
</thead>
<tbody>
<tr>
<td>HKUST</td>
<td>Wong</td>
<td>Au</td>
</tr>
<tr>
<td>HKUST</td>
<td>Chin</td>
<td>Au</td>
</tr>
<tr>
<td>Central</td>
<td>Wong</td>
<td>Jones</td>
</tr>
<tr>
<td>Central</td>
<td>null</td>
<td>Cheng</td>
</tr>
</tbody>
</table>

• A 3NF table still has problems
  - redundancy (e.g., we repeat that Au works at HKUST branch)
  - need to use null values (e.g., to represent that Cheng works at Central even though he is not assigned any customers).
The initial table $R$ and its FDs $F$

Output: The table decomposition $S$ into 3NF

Compute the canonical cover $F_c$ of $F$

$S = \emptyset$

for each FD $X \rightarrow Y$ in the canonical cover $F_c$

Create table $T = (X, Y)$

$S = S \cup \{T\}$

if no scheme in $S$ contains a candidate key for $R$

Choose any candidate key $CN$

$S = S \cup \{\text{table with attributes of } CN\}$

This algorithm always creates a lossless-join, dependency-preserving, 3NF decomposition
• Initial table:
  - Bank=(branch-name, customer-name, banker-name, office-number)

• Functional dependencies (also canonical cover):
  - \{banker-name\}→\{branch-name, office-number\}
  - \{customer-name, branch-name\}→\{banker-name\}

• Candidate Keys:
  - \{customer-name, branch-name\} or \{customer-name, banker-name\}

• \{banker-name\}→\{branch-name, office-number\} violates 3NF

• 3NF tables: – for each FD in the canonical cover create a table
  - Banker = (banker-name, branch-name, office-number)
  - Customer-Branch = (customer-name, branch-name, banker-name)

• Since Customer-Branch contains a candidate key for Bank, we are done

• Question: is the decomposition lossless and dependency preserving?
  Answer: Yes – all decompositions generated by this algorithm have these properties