Schema Design & Refinement (aka Normalization)
“Crouching Beer, Hidden Bratwurst” Team:

Motivation

Students(ssn, name, age, can-drink-beer)
Courses(cid, name)
Takes(ssn, cid)
Example of What is Wrong

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>oh yes</td>
</tr>
</tbody>
</table>

There is redundancy here, leading to all kinds of problems (also called anomalies)

update anomalies = update one item and forget the others
deletion anomalies = delete multiple items
  if delete all, then loose information

some other anomalies too
A solution: refine this table by breaking it down into two tables

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>18</td>
<td>yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
A solution: refine this table by breaking it down into two tables

So instead of

Students(ssn, name, age, can-drink)
Courses(cid, name)
Take(ssn,cid)

We will have

Students(ssn, name, age)
Drink-ability(age, can-drink)
Courses(cid, name)
Take(ssn, cid)
Need a general solution that works on any relational schema

• Intuition
  – given an ER diagram
  – translate it into a relational schema $R$
  – think about all dependency constraints that can apply to $R$
    • such as “age determines can-drink-beer”
  – use these constraints to detect if $R$ is a bad schema
    • such as having some kind of redundancy
  – then refine $R$ into a schema $R^*$ with less redundancy
In practice, dependencies such as age $\rightarrow$ can-drink are called “functional dependencies”.

We need to first formalize and study (1) functional dependencies, and (2) keys for tables before we can talk about (1) how to detect bad tables, and (2) how to break them down.
Functional Dependencies

• A form of constraint (hence, part of the schema)
• Finding them is part of the database design
• Used heavily in schema refinement

Definition:

If two tuples agree on the attributes

\[ A_1, A_2, \ldots, A_n \]

then they must also agree on the attributes

\[ B_1, B_2, \ldots, B_m \]

Formally:

\[ A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m \]
## Examples

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>oh yes</td>
</tr>
</tbody>
</table>

SSN ➔ Name, Age, Can-Drink  
Age ➔ Can-Drink  
SSN, Age ➔ Name, Can-Drink
Examples

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E1847</td>
<td>John</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>lawyer</td>
</tr>
</tbody>
</table>

- EmpID  → Name, Phone, Position
- Position  → Phone
- but Phone  ← Position
How Do We Infer FDs?

• Create ER Diagram
• Translate into a relational schema
• Think hard about what FDs are valid for that relational schema
  – think from an application point of view

• An FD is an inherent property of an application
• It is not something we can infer from a set of tuples
How Do We Infer FDs?

• Given a table with a set of tuples
  – the best we can do is confirm that a FD seems to be valid
  – or to infer that a FD is definitely invalid
  – we can never prove that a FD is valid
In General

• To confirm $A \rightarrow B$, erase all other columns

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th></th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>X1</td>
<td></td>
<td>Y1</td>
<td></td>
</tr>
<tr>
<td>X2</td>
<td></td>
<td>Y2</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

• check if the remaining relation is many-one
  – if yes, then the FD is probably valid
  – if no, then the FD is definitely invalid
### Example: Position ➔ Phone

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E1847</td>
<td>John</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>lawyer</td>
</tr>
</tbody>
</table>

How about Name ➔ Phone?
Keys

• Key of a relation R is a set of attributes that
  – functionally determines all attributes of R
  – none of its subsets determines all attributes of R

• Superkey
  – a set of attributes that contains a key

• We will need to know the keys of the relations in a DB schema, so that we can refine the schema
Finding the Keys of a Relation

Given a relation constructed from an E/R diagram, what is its key?

Rules:
1. If the relation comes from an entity set, the key of the relation is the set of attributes which is the key of the entity set.
Finding the Keys

Rules:
2. If the relation comes from a many-many relationship, the key of the relation is the set of all attribute keys in the relations corresponding to the entity sets

\[ \text{buys}(\text{name, ssn, date}) \]
Finding the Keys

But: if there is an arrow from the relationship to E, then we don’t need the key of E as part of the relation key.

Purchase(name, sname, ssn, card-no)
Finding the Keys

More rules:
• Many-one, one-many, one-one relationships
• Multi-way relationships
• Weak entity sets

(Try to find them yourself)
Why specifying keys and FDs?

• Why keys?
  – help identify entities/tuples
  – imply certain FDs

• Why FDs?
  – give us more integrity constraints for the application

• More importantly
  – having keys and FDs will help us detect that a table is “bad”, and helps us determine how to decompose the table
An Example

Students(ssn, name, age, can-drink)
Courses(cid, name)
Takes(ssn, cid)

So what are the FDs inferred from keys?
ssn $\rightarrow$ ...
cid $\rightarrow$ ...

We also add
age $\rightarrow$ can-drink
Once the team has specified some keys and FDs, we can’t just stop there

• We want to infer **all FDs** that may be logically implied
  – e.g., if team says $A \rightarrow B$, $B \rightarrow C$, then we also have $A \rightarrow C$

• Given a set of attributes, we also want to infer **all attributes** that are functionally determined by these given attributes

• Knowing these will help us detect if a table is bad and how to decompose it
Inferring All FDs

• Given a relation schema R & a set S of FDs
  – is the FD f logically implied by S?

• Example
  – R = {A,B,C,G,H,I}
  – S = A → B, A → C, CG → H, CG → I, B → H
  – would A → H be logically implied?
  – yes (you can prove this, using the definition of FD)

• Closure of S: S+ = all FDs logically implied by S

• How to compute S+?
  – we can use Armstrong's axioms
Armstrong's Axioms

• Reflexivity rule
  – $A_1A_2...A_n \Rightarrow$ a subset of $A_1A_2...A_n$

• Augmentation rule
  – $A_1A_2...A_n \Rightarrow B_1B_2...B_m$, then
    $A_1A_2...A_n C_1C_2..C_k \Rightarrow B_1B_2...B_m C_1C_2...C_k$

• Transitivity rule
  – $A_1A_2...A_n \Rightarrow B_1B_2...B_m$ and
    $B_1B_2...B_m \Rightarrow C_1C_2...C_k$, then
    $A_1A_2...A_n \Rightarrow C_1C_2...C_k$
Inferring $S^+$ using Armstrong's Axioms

- $S^+ = S$
- Loop
  - foreach $f$ in $S$, apply reflexivity and augment. rules
  - add the new FDs to $S^+$
  - foreach pair of FDs in $S$, apply the transitivity rule
  - add the new FD to $S^+$
- Until $S^+$ does not change any further
- Basically, just apply rules until can’t apply anymore
Additional Rules

- **Union rule**
  - $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$
  - $(X, Y, Z$ are sets of attributes$)$

- **Decomposition rule**
  - $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

- **Pseudo-transitivity rule**
  - $X \rightarrow Y$ and $YZ \rightarrow U$, then $XZ \rightarrow U$

- **These rules can be inferred from Armstrong's axioms**
Find All Attributes that are Functionally Determined by a Set of Attributes

Given a set of attributes \{A_1, \ldots, A_n\} and a set of dependencies \(S\).

Problem: find all attributes \(B\) such that:

any relation which satisfies \(S\) also satisfies:

\[ A_1, \ldots, A_n \rightarrow B \]

That is, all attributes \(B\) that are functionally determined by the \(A_i\)

The closure of \(\{A_1, \ldots, A_n\}\) is the set of all such attributes \(B\)
Algorithm to Compute Closure

Start with $X=\{A_1, \ldots, A_n\}$.

**Repeat until $X$ doesn’t change** do:

- if $B_1, B_2, \ldots, B_n \rightarrow C$ is in $S$, and $B_1, B_2, \ldots, B_n$ are all in $X$, and $C$ is not in $X$

  then

  add $C$ to $X$.

Just apply FDs until can’t apply anymore
Example

\[
\begin{align*}
A & \quad B \quad \rightarrow \quad C \\
A & \quad D \quad \rightarrow \quad E \\
B & \quad \rightarrow \quad D \\
A & \quad F \quad \rightarrow \quad B
\end{align*}
\]

Closure of \(\{A, B\}\): \(X = \{A, B, C, D, E\}\)

Closure of \(\{A, F\}\): \(X = \{A, F, B, D, C, E\}\)
Usage for Attribute Closure

- Test if $X$ is a superkey
  - compute $X^+$, and check if $X^+$ contains all attrs of $R$
- Check if $X \Rightarrow Y$ holds
  - by checking if $Y$ is contained in $X^+$
- Another way to compute closure $S^+$ of FDs
  - for each subset of attributes $X$ in relation $R$, compute $X^+$
  - for each subset of attributes $Y$ in $X^+$, output the FD $X \Rightarrow Y$
Review

• We have learned about keys and FDs
• We have learned about how to reason with them
  – given a set of FDs, infer all new applicable FDs
  – given a set of attributes X, infer all new attributes that are functionally determined by X
• Now we will look at how to use them to detect that a table is “bad”.
• We say a table is “bad” if it is not in Boyce-Codd normal form
A relation $R$ is in BCNF if and only if:

Whenever there is a nontrivial $\text{FD} \ A_1, A_2 \ldots A_n \rightarrow B$ for $R$, it is the case that $\{ A_1, A_2 \ldots A_n \}$ is a super-key for $R$. 

Boyce-Codd Normal Form
Example: This is not in BCNF

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>oh yes</td>
</tr>
</tbody>
</table>

ssn $\rightarrow$ name, age, can-drink

age $\rightarrow$ can-drink

ssn is a key

For each FD $A \rightarrow B$, ask: is $A$ a superkey?
If not, then the FD violates BCNF, relation is not in BCNF
To do so: (a) from current set of FDs, infer all FDs
(b) find the closure of $A$
Example in BCNF

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>18</td>
<td>yes</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

ssn $\rightarrow$ name, age

age $\rightarrow$ can-drink

Any relation of only two attributes is in BCNF
Example of non-BCNF

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>123-321-99</td>
<td>(201) 555-1234</td>
</tr>
<tr>
<td>Fred</td>
<td>123-321-99</td>
<td>(206) 572-4312</td>
</tr>
<tr>
<td>Joe</td>
<td>909-438-44</td>
<td>(908) 464-0028</td>
</tr>
<tr>
<td>Joe</td>
<td>909-438-44</td>
<td>(212) 555-4000</td>
</tr>
</tbody>
</table>

What are the dependencies?
- **SSN** → **Name**

What are the keys?

Is it in BCNF?
Example of BCNF

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-321-99</td>
<td>Fred</td>
</tr>
<tr>
<td>909-438-44</td>
<td>Joe</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Phone Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-321-99</td>
<td>(201) 555-1234</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 572-4312</td>
</tr>
<tr>
<td>909-438-44</td>
<td>(908) 464-0028</td>
</tr>
<tr>
<td>909-438-44</td>
<td>(212) 555-4000</td>
</tr>
</tbody>
</table>
What About This?

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>gadgets</td>
</tr>
<tr>
<td>OneClick</td>
<td>$24.99</td>
<td>camera</td>
</tr>
</tbody>
</table>
How to Detect that a Table is not in BCNF?

A relation R is in BCNF if and only if:

Whenever there is a nontrivial FD for R, it is the case that \( \{ A_1, A_2, \ldots, A_n \} \) is a super-key for R.

So we start by creating the ER diagram, specifying keys
Then translate it into relational tables, specifying keys
Then add as many FDs as we can think of
Then infer all other FDs
Then for each FD X \( \rightarrow Y \), check if X is a superkey
(a key is also a superkey); one way to do this is to compute the closure of X
Once we know that a table is not in BCNF, how do we decompose it?
BCNF Decomposition

Find a dependency that violates the BCNF condition:

\[ A_1, A_2, \ldots, A_n \rightarrow B_1, B_2, \ldots, B_m \]

Heuristics: expand \( B_1, B_2, \ldots, B_m \) “as much as possible”

Decompose:

- \( R1 \): B’s
- \( R2 \): A’s and remaining attributes

Continue until there are no BCNF violations left.

Any 2-attribute relation is in BCNF.
Decompose into BCNF

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>oh yes</td>
</tr>
</tbody>
</table>

SSN ➔ name, age, can-drink
age ➔ can-drink

<table>
<thead>
<tr>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>18</td>
<td>yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

age ➔ can-drink

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
</tr>
</tbody>
</table>

ssn ➔ name, age
Example Decomposition

Person:

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>Age</th>
<th>EyeColor</th>
<th>PhoneNumber</th>
</tr>
</thead>
</table>

Functional dependencies:

SSN → Name, Age, Eye Color

BNCF: R1(SSN, Name, Age, EyeColor),
      R2(SSN, PhoneNumber)
Another Example

• Persons(SSN, name, age, eye-color, phone, can-drink)
  – SSN $\Rightarrow$ name, age, eye-color
  – age $\Rightarrow$ can-drink

• What we can infer from the above?
  – SSN $\Rightarrow$ name, age, eye-color, can-drink
  – SSN is NOT a key nor a superkey
  – not in BCNF

• Decomposing
  – use SSN $\Rightarrow$ name, age, eye-color, can-drink (biggest expansion)
  – R1(SSN, name, age, eye-color, can-drink)
  – R2(SSN, phone)
Another Example

• Decomposing
  – use SSN $\rightarrow$ name, age, eye-color, can-drink
  – R1(SSN, name, age, eye-color, can-drink)
    SSN $\rightarrow$ name, age, eye-color, can-drink
    age $\rightarrow$ can-drink
  – R2(SSN, phone)

• Need to decompose R1, using age $\rightarrow$ can-drink
  – R3(age, can-drink)
    age $\rightarrow$ can-drink
  – R4(age, SSN, name, eye-color)
    SSN $\rightarrow$ age, name, eye-color
  – R2(SSN, phone)
We have learned (a) how to detect that a table is not in BCNF, (b) how to decompose it.

How do we know that this decomposition is a good one? What do we mean by “good” here?
Desirable Properties of Schema Decomposition (that is, Schema Refinement)

1) minimize redundancy
2) avoid info loss
3) preserve dependency
4) ensure good query performance
Decompositions in General

Let $R$ be a relation with attributes $A_1, A_2, \ldots, A_n$

Create two relations $R1$ and $R2$ with attributes

$$B_1, B_2, \ldots, B_m \quad C_1, C_2, \ldots, C_l$$

Such that:

$$B_1, B_2, \ldots, B_m \cup C_1, C_2, \ldots, C_l = A_1, A_2, \ldots, A_n$$

And

-- $R1$ is the projection of $R$ on $B_1, B_2, \ldots, B_m$

-- $R2$ is the projection of $R$ on $C_1, C_2, \ldots, C_l$
## Example

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>oh yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>18</td>
<td>yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Desirable Property #1: Minimize redundancy

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>yes</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>18</td>
<td>yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Certain Decomposition May Cause Info Loss

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>DoubleClick</td>
<td>29.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>

Decompose on: **Name, Category** and **Price, Category**

When we put it back:

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>Gadget</td>
</tr>
<tr>
<td>OneClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td><strong>OneClick</strong></td>
<td><strong>29.99</strong></td>
<td><strong>Camera</strong></td>
</tr>
<tr>
<td>DoubleClick</td>
<td>24.99</td>
<td>Camera</td>
</tr>
<tr>
<td>DoubleClick</td>
<td>29.99</td>
<td>Camera</td>
</tr>
</tbody>
</table>

Cannot recover information
Lossless Decompositions

A decomposition is *lossless* if we can recover:

\[ R(A, B, C) \]

\[ R_1(A, B) \quad R_2(A, C) \]

\[ R'(A, B, C) \quad \text{should be the same as} \quad R(A, B, C) \]

\[ R' \text{ is in general larger than } R. \quad \text{Must ensure } R' = R \]
Put Another Way: "Lossless" Joins

• The main idea: if you decompose a relation schema, then join the parts of an instance via a natural join, you might get more rows than you started with, i.e., spurious tuples
  – This is bad!
  – Called a "lossy join".

• Goal: decompositions which produce only "lossless" joins
  – "non-additive" join is more descriptive
  – because we don’t want to add more tuples

• Desirable Property #2: Lossless decomposition
Dependency Preserving

• Given a relation $R$ and a set of FDs $S$
• Suppose we decompose $R$ into $R_1$ and $R_2$
• Suppose
  – $R_1$ has a set of FDs $S_1$
  – $R_2$ has a set of FDs $S_2$
  – $S_1$ and $S_2$ are computed from $S$
• We say the decomposition is dependency preserving if by enforcing $S_1$ over $R_1$ and $S_2$ over $R_2$, we can enforce $S$ over $R$
### Example

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
<td>oh yes</td>
</tr>
</tbody>
</table>

SSN $\Rightarrow$ name, age, can-drink

age $\Rightarrow$ can-drink

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Age</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Dave</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>Mike</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Jane</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Liu</td>
<td>19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Age</th>
<th>Can-Drink</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>no</td>
</tr>
<tr>
<td>17</td>
<td>no</td>
</tr>
<tr>
<td>18</td>
<td>yes</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

ssn $\Rightarrow$ name, age

age $\Rightarrow$ can-drink
Another Example

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
</table>

FD’s: $\text{Unit} \rightarrow \text{Company}$; $\text{Company, Product} \rightarrow \text{Unit}$

Consider the decomposition:

- $\text{Unit} \rightarrow \text{Company}$
- No FDs
So What’s the Problem?

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaga99</td>
<td>UW</td>
<td>databases</td>
</tr>
<tr>
<td>Bingo</td>
<td>UW</td>
<td>databases</td>
</tr>
</tbody>
</table>

No problem so far. All *local* FD’s are satisfied.

Let’s put all the data back into a single table again:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Galaga99</td>
<td>UW</td>
<td>databases</td>
</tr>
<tr>
<td>Bingo</td>
<td>UW</td>
<td>databases</td>
</tr>
</tbody>
</table>

Violates the dependency: *company, product* -> *unit*!
Preserving FDs

• Such a decomposition is not “dependency-preserving.”

• Desirable Property #3: always have FD-preserving decompositions

• We will talk about "Desirable Property #4: Ensure Good Query Performance" later
Review

• When decomposing a relation $R$, we want the decomposition to
  – minimize redundancy
  – avoid info loss
  – preserve dependencies (i.e., constraints)
  – ensure good query performance
• These objectives can be conflicting
• Boyce-Codd normal form achieves some of these
In particular

- BCNF removes certain types of redundancy
- For examples of redundancy that it cannot remove, see "multivalued redundancy"
- BCNF avoids info loss
- BCNF is not always dependency preserving
Recall: Lossless Decompositions

A decomposition is *lossless* if we can recover:

\[ R(A,B,C) \]

\[
\{ R_1(A,B) , R_2(A,C) \}
\]

\[ R'(A,B,C) = R(A,B,C) \]

\( R' \) is in general larger than \( R \). Must ensure \( R' = R \)
Decomposition Based on BCNF is Necessarily Lossless

\[ R(A, B, C), \quad A \to C \]

BCNF: \( R1(A, B), \quad R2(A, C) \)

Some tuple \((a,b,c)\) in \(R\) decomposes into \((a,b)\) in \(R1\) and \((a,c)\) in \(R2\).

\((a,b',c')\) also in \(R\) \((a,b')\) also in \(R1\) \((a,c')\) also in \(R2\)

Recover tuples in \(R\): \((a,b,c)\), \((a,b,c')\), \((a,b',c)\), \((a,b',c')\) also in \(R\)?

Can \((a,b,c')\) be a bogus tuple? What about \((a,b',c')\)?

61
However,

- BCNF is not always dependency preserving
- In fact, some times we cannot find a BCNF decomposition that is dependency preserving
An Example

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
</table>

FD’s: Unit $\rightarrow$ Company; Company, Product $\rightarrow$ Unit

Consider the decomposition:

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Unit $\rightarrow$ Company</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Unit</th>
<th>Product</th>
<th>No FDs</th>
</tr>
</thead>
</table>
BCNF is called a “normal form”. Many other types of normal forms exist.

First Normal Form = all attributes are atomic
Second Normal Form (2NF) = old and obsolete

Boyce Codd Normal Form (BCNF)
Third Normal Form (3NF)
Fourth Normal Form (4NF)

Others...
A simple condition for removing anomalies from relations:

A relation R is in 3rd normal form if:

Whenever there is a nontrivial dependency $A_1, A_2, ..., A_n \rightarrow B$ for $R$, then \{$A_1, A_2, ..., A_n$\} is a super-key for $R$, or $B$ is part of a key.
## An Example

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
</table>

FD’s: $\text{Unit} \rightarrow \text{Company}$; $\text{Company, Product} \rightarrow \text{Unit}$
3NF (General Definition)

• A relation is in Third Normal Form (3NF) if whenever $X \rightarrow A$ holds, either $X$ is a superkey, or $A$ is a prime attribute.

Informally: everything depends on the key or is in the key.

• Despite the thorny technical definitions that lead up to it, 3NF is intuitive and not hard to achieve. Aim for it in all designs unless you have strong reasons otherwise.
3NF vs. BCNF

- R is in BFNC if whenever \( X \rightarrow A \) holds, then \( X \) is a superkey.
- Slightly stronger than 3NF.
- Example: \( R(A,B,C) \) with \( \{A,B\} \rightarrow C, \ C \rightarrow A \)
  - 3NF but not BCNF

Guideline: Aim for BCNF and settle for 3NF
Decomposing $R$ into 3NF

- The algorithm is complicated
- 1. Get a “minimal cover” of FDs
- 2. Find a lossless-join decomposition of $R$ (which might miss dependencies)
- 3. Add additional relations to the decomposition to cover any missing FDs of the cover
- Result will be lossless, will be dependency-preserving 3NF; might not be BCNF
Normal Forms

First Normal Form = all attributes are atomic
Second Normal Form (2NF) = old and obsolete

Boyce Codd Normal Form (BCNF)
Third Normal Form (3NF)
Fourth Normal Form (4NF)

Others...
## Multi-valued Dependencies

<table>
<thead>
<tr>
<th>SSN</th>
<th>Phone Number</th>
<th>Course</th>
</tr>
</thead>
<tbody>
<tr>
<td>123-321-99</td>
<td>(206) 572-4312</td>
<td>CSE-444</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 572-4312</td>
<td>CSE-341</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 432-8954</td>
<td>CSE-444</td>
</tr>
<tr>
<td>123-321-99</td>
<td>(206) 432-8954</td>
<td>CSE-341</td>
</tr>
</tbody>
</table>

The multi-valued dependencies are:

- SSN → Phone Number
- SSN → Course
Definition of Multi-valued Dependency

Given $R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_p)$

the MVD $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$ holds if:

for any values of $A_1, \ldots, A_n$
the “set of values” of $B_1, \ldots, B_m$
is “independent” of those of $C_1, \ldots, C_p$
Definition of MVDs Continued

Equivalently: the decomposition into

\[ R_1(A_1, \ldots, A_n, B_1, \ldots, B_m), \quad R_2(A_1, \ldots, A_n, C_1, \ldots, C_p) \]

is lossless

Note: an MVD \( A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m \)
Implicitly talks about “the other” attributes \( C_1, \ldots, C_p \)
Rules for MVDs

If $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$

then $A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$

Other rules in the book
**4th Normal Form (4NF)**

$R$ is in 4NF if whenever:

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m$$

is a nontrivial MVD,

then $A_1, \ldots, A_n$ is a superkey

*Same as BCNF with FDs replaced by MVDs*
Multivalued Dependencies (MVDs)

• $X \rightarrow\rightarrow Y$ means that given $X$, there is a unique set of possible $Y$ values (which do not depend on other attributes of the relation)

• PARENTNAME $\rightarrow\rightarrow$ CHILDNAME

• An FD is also a MVD

• MVD problems arise if there are two independent 1:N relationships in a relation.
Confused by Normal Forms?

In practice: (1) 3NF is enough, (2) don’t overdo it!
Normal Forms

First Normal Form = all attributes are atomic
Second Normal Form (2NF) = old and obsolete

Boyce Codd Normal Form (BCNF)
Third Normal Form (3NF)
Fourth Normal Form (4NF)

Others...
Fifth Normal Form

• Sometimes a relation cannot be losslessly decomposed into two relations, but can be into three or more.

• 5NF captures the idea that a relation scheme must have some particular lossless decomposition ("join dependency").

• Finding actual 5NF cases is difficult.
Normalization Summary

- 1NF: usually part of the woodwork
- 2NF: usually skipped
- 3NF: a biggie
  - always aim for this
- BCNF and 4NF: tradeoffs start here
  - in re: d-preserving and losslessness
- 5NF: You can say you've heard of it...
Caveat

• Normalization is not the be-all and end-all of DB design

• Example: suppose attributes A and B are always used together, but normalization theory says they should be in different tables.
  – decomposition might produce unacceptable performance loss (extra disk reads)

• Desirable Property #4: Good query performance

• Plus -- there are constraints other than FDs and MVDs
Current Trends

• Data Warehouses
  – huge historical databases, seldom or never updated after creation
  – joins expensive or impractical
  – argues against normalization

• Everyday relational DBs
  – aim for BCNF, settle for 3NF
Relational Schema Design (or Logical Design)

Conceptual Model:

Relational Model:
- create tables
- specify FD’s
- find keys

Normalization
- use FDs to decompose tables to achieve better design