Part II:
Using FD Theory to do Database Design
Recall that poorly designed table?

<table>
<thead>
<tr>
<th>part</th>
<th>manufacturer</th>
<th>manAddress</th>
<th>seller</th>
<th>sellerAddress</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1983</td>
<td>Hammers ’R Us</td>
<td>99 Pinecrest</td>
<td>ABC</td>
<td>1229 Bloor W</td>
<td>5.59</td>
</tr>
<tr>
<td>8624</td>
<td>Lee Valley</td>
<td>102 Vaughn</td>
<td>ABC</td>
<td>1229 Bloor W</td>
<td>23.99</td>
</tr>
<tr>
<td>9141</td>
<td>Hammers ’R Us</td>
<td>99 Pinecrest</td>
<td>ABC</td>
<td>1229 Bloor W</td>
<td>12.50</td>
</tr>
<tr>
<td>1983</td>
<td>Hammers ’R Us</td>
<td>99 Pinecrest</td>
<td>Walmart</td>
<td>5289 St Clair W</td>
<td>4.99</td>
</tr>
</tbody>
</table>

- We can now express the relationships as FDs:
  - part → manufacturer
  - manufacturer → address
  - seller → address

- The FDs tell us there can be redundancy, thus the design is bad.
- That’s why we care about FDs.
Decomposition

To improve a badly-designed schema \( R(A_1, \ldots, A_n) \), we will decompose it into smaller relations

\[ R_1(B_1, \ldots, B_j) \text{ and } R_2(C_1, \ldots, C_k) \text{ such that:} \]

\( R_1 = \pi_{B_1, \ldots, B_j}(R) \)

\( R_2 = \pi_{C_1, \ldots, C_k}(R) \)

\( \{B_1, \ldots, B_j\} \cup \{C_1, \ldots, C_k\} = \{A_1, \ldots, A_n\} \)

\( R_1 \bowtie R_2 = R \)
R(A₁, ... Aₙ)  

Set of attributes: A

Decompose into:

- R₁(B₁, ... Bⱼ)  
  
  Set of attributes: B, and

- R₂(C₁, ... Cₖ)  
  
  Set of attributes: C

\[ B \cup C = A, \quad R₁ \bowtie R₂ = R \]

\[
R₁ = \pi_B(R) \\
R₂ = \pi_C(R)
\]
But *which* decomposition?

- Decomposition can definitely improve a schema.
- But which decomposition? There are many possibilities.
- And how can we be sure a new schema doesn’t exhibit other anomalies?
- Boyce-Codd Normal Form *guarantees* it.
Boyce-Codd Normal Form

◆ We say a relation $R$ is in $BCNF$ if for every nontrivial FD $X \rightarrow Y$ that holds in $R$, $X$ is a superkey.

❖ Remember: *nontrivial* means $Y$ is not contained in $X$.

❖ Remember: a *superkey* doesn’t have to be minimal.

◆ [Exercise]
Intuition

In other words, BCNF requires that:

Only things that functionally determine *everything* can functionally determine *anything*.

Why is the BCNF property valuable?

Note:

- FDs are not the problem. They are facts!
- The schema (in the context of the FDs) is the problem.
BCNF_decomp\((R, F)\):

If an FD \( X \rightarrow Y \) in \( F \) violates BCNF

Compute \( X^+ \).

Replace \( R \) by two relations with schemas:

\[
R_1 = X^+ \\
R_2 = R - (X^+ - X)
\]

Project the FD’s \( F \) onto \( R_1 \) and \( R_2 \).

Recursively decompose \( R_1 \) and \( R_2 \) into BCNF.

[Example]
1) Start with the LHS of the violating FD.
2) Close the LHS to get one new relation
3) Everything except the new stuff is the other new relation. $X$ is in both new relations to make a connection between them.
Some comments on BCNF decomp

◆ If more than one FD violates BCNF, you may decompose based on any one of them.
  ▪ So there may be multiple results possible.
◆ The new relations we create may not be in BCNF. We must recurse.
  ▪ We only keep the relations at the “leaves”.
◆ How does the decomposition step help? [Exercise]
Speed-ups for BCNF decomposition

- Don’t need to know any keys.
  - Only superkeys matter.
- And don’t need to know all superkeys.
  - Only need to check whether the LHS of each FD is a superkey.
  - Use the closure test (simple and fast!).
BCNF

Every attribute depends on:

- The key
- The whole key
- And nothing but the key...

so help me Codd....
More speed-ups

◆ When projecting FDs onto a new relation, check each new FD:
  ◦ Does the new relation violate BCNF because of this FD?
◆ If so, abort the projection.
  ◦ You are about to discard this relation anyway (and decompose further).
Properties of Decompositions
What we want from a decomposition

1. **No anomalies.**

2. **Lossless Join** : It should be possible to
   a) project the original relations onto the decomposed schema
   b) then reconstruct the original by joining. We should get back exactly the original tuples.

3. **Dependency Preservation** :
   All the original FD’s should be satisfied.
What is lost in a “lossy” join?

◆ For any decomposition, it is the case that:
  ▶ r ⊆ r₁ ⋈ ... ⋈ rₙ
  ▶ I.e., we will get back every tuple.
◆ But it may not be the case that:
  ▶ r ⊇ r₁ ⋈ ... ⋈ r
  ▶ I.e., we can get spurious tuples.
◆ [Exercise]
What BCNF decomposition offers

1. **No anomalies**: ✓ (Due to no redundancy)
2. **Lossless Join**: ✓ (Section 3.4.1 argues this)
3. **Dependency Preservation**: ✗
The BCNF *property* does not guarantee lossless join

◆ If you use the BCNF decomposition algorithm, a lossless join is guaranteed.
◆ If you generate a decomposition some other way
  ▶ you have to check to make sure you have a lossless join
  ▶ even if your schema satisfies BCNF!
◆ We’ll learn an algorithm for this check later.
Preservation of dependencies

◆ BCNF decomposition does not guarantee preservation of dependencies.
◆ I.e., in the schema that results, it may be possible to create an instance that:
  ◦ satisfies all the FDs in the final schema,
  ◦ but violates one of the original FDs.
◆ Why? Because the algorithm goes too far — breaks relations down too much.
◆ [Exercise]
3NF is less strict than BCNF

◆ 3rd Normal Form (3NF) modifies the BCNF condition to be less strict.
◆ An attribute is *prime* if it is a member of any key.
◆ $X \rightarrow A$ violates 3NF iff
  $X$ is not a superkey and $A$ is not prime.
◆ I.e., it’s ok if $X$ is not a superkey as long as $A$ is prime.
◆ [Exercise]
F is a set of FDs; L is a set of attributes. Synthesize and return a schema in 3\textsuperscript{rd} Normal Form.

\textbf{3NF\_synthesis}(F, L):

Construct a minimal basis M for F.

For each FD $X \rightarrow Y$ in M

Define a new relation with schema $X \cup Y$.

If no relation is a superkey for L

Add a relation whose schema is some key.

[Example]
3NF synthesis doesn’t “go too far”

◆ BCNF decomposition doesn’t stop decomposing until in all relations:
  ▶ if $X \rightarrow A$ then $X$ is a superkey.

◆ 3NF generates relations where:
  ▶ $X \rightarrow A$ and yet $X$ is not a superkey, but $A$ is at least prime.

◆ [Example]
What a 3NF decomposition offers

1. **No anomalies** : ✗
2. **Lossless Join** : ✓
3. **Dependency Preservation** : ✓

◆ Neither BCNF nor 3NF can guarantee all three! We must be satisfied with 2 of 3.
◆ Decompose too far ⇒ can’t enforce all FDs.
◆ Not far enough ⇒ can have redundancy.
◆ We consider a schema “good” if it is in either BCNF or 3NF.
How can we get anomalies?

◆ 3NF synthesis guarantees that the resulting schema will be in 3rd normal form.
◆ This allows FDs with a non-superkey on the LHS.
◆ This allows redundancy, and thus anomalies.
How do we know...?

... that the algorithm guarantees:

- **3NF**: A property of minimal bases [see the textbook for more]
- **Preservation of dependencies**: Each FD from a minimal basis is contained in a relation, thus preserved.
- **Lossless join**: We’ll return to this once we know how to test for lossless join.
“Synthesis” vs “decomposition”

◆ 3NF synthesis:
  ▶ We build up the relations in the schema from nothing.

◆ BCNF decomposition:
  ▶ We start with a bad relation schema and break it down.
Testing for a Lossless Join

◆ If we project $R$ onto $R_1, R_2, \ldots, R_k$, can we recover $R$ by rejoining?

◆ We will get all of $R$.
  
  ◦ Any tuple in $R$ can be recovered from its projected fragments. This is guaranteed.

◆ But will we get only $R$?
  
  ◦ Can we get a tuple we didn’t have in $R$? This part we must check.
Aside: when we don’t need to test for lossless Join

- Both BCNF decomposition and 3NF synthesis guarantee lossless join.
- So we don’t need to test for lossless join if the schema was generated via BCNF decomposition or 3NF synthesis.
- But merely satisfying BCNF or 3NF does not guarantee a lossless join!
The Chase Test

- Suppose tuple \( t \) appears in the join.
- Then \( t \) is the join of projections of some tuples of \( R \), one for each \( R_i \) of the decomposition.
- Can we use the given FD’s to show that one of these tuples must be \( t \)?
- [Example]
Setup for the Chase Test

- Start by assuming $t = abc...$.
- For each $i$, there is a tuple $s_i$ of $R$ that has $a, b, c,...$ in the attributes of $R_i$.
- $s_i$ can have any values in other attributes.
- We’ll use the same letter as in $t$, but with a subscript, for these components.
The algorithm

1. If two rows agree in the left side of a FD, make their right sides agree too.

2. Always replace a subscripted symbol by the corresponding unsubscripted one, if possible.

3. If we ever get a completely unsubscripted row, we know any tuple in the project-join is in the original (i.e., the join is lossless).

4. Otherwise, the final tableau is a counterexample (i.e., the join is lossy).

[Exercise]