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 → manAddress
  ◆ seller → sellerAddress

◆ The FDs tell us there can be redundancy, thus the design is bad.

◆ That’s why we care about FDs.

◆ [Exercise 1]
Decomposition

To improve a badly-designed schema \( R(A_1, A_2, \ldots, A_n) \), we will decompose it into smaller relations \( S(B_1, B_2, \ldots, B_m) \) and \( T(C_1, C_2, \ldots C_k) \) such that:

\[
S = \pi_{B_1, B_2, \ldots, B_m}(R) \\
T = \pi_{C_1, C_2, \ldots, C_k}(R) \\
\{A_1, A_2, \ldots, A_n\} = \{B_1, B_2, \ldots, B_m\} \cup \{C_1, C_2, \ldots C_k\}
\]
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 (BCNF)** 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: $X$ and $Y$ are sets (could be singletons too, though)
- Remember: nontrivial means $Y$ is not contained in $X$.
- Remember: a superkey doesn’t have to be minimal.
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.

![Diagram showing a relation R with attributes X and Y, violating BCNF](image)

Not in BCNF!
Or
“Violates BCNF”
Example 1

◆ Person (SIN#, Name, Address)
  FDs = {SIN# -> Name, Address}

◆ Is this relation in BCNF?
  ♦ Yes, because SIN# is a superkey (in fact, a key).
  ♦ SIN#⁺ = SIN#, Name, Address
Example 2

- **Person** (SIN#, Name, Address, Hobby)
  
  \[ \text{FDs} = \{ \text{SIN#} \rightarrow \text{Name, Address} \} \]

- **Is this relation in BCNF?**
  - No, because SIN# is not a key.
  - What about if we had
    
    \[ \text{FDs} = \{ \text{SIN#, Hobby} \rightarrow \text{Name, Address} \} \]
  - Yes, LHS is a key now.
More complex example

- Relation: Books (Author, Nationality, BookTitle, Genre, NumberOfPages, Rating)
- FDs = {Author -> Nationality, BookTitle -> Genre, NumberOfPages, Rating}

Is this in BCNF?
- No. The only key is (Author, BookTitle)

How would you intuitively make it be?
- Decompose based on each LHS’s closure
Intuition

In other words, BCNF requires that:

Only things that FD *everything*
can FD 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.
1) Start with the LHS of the violating FD.

2) Close the LHS to get one new relation $R_1$

$R_1 = X^+$

3) Everything except the new stuff is the other new relation $R_2$.

$R_2 = R - X^+ + X$

$X$ is in both new relations to make a connection between them.
Example

- FoodTasters (name, addr, foodLiked, producer, favFood)
- \{a. name->addr, b. name->favFood, c. foodLiked->producer\}

- Pick FD that violates BCNF: name -> addr
  name\(^+\) = name, addr, favFood

- Decomposed relations and projected FDs:
  - FoodTasters1 (name, addr, favFood) \{name->addr, name->favFood\}
  - FoodTasters2 (name, foodLiked, producer) \{foodLiked->producer\}
Example

- Decomposed relations and projected FDs:
  - FoodTasters1 (name, addr, favFood) \{name->addr, name->favFood\}
  - FoodTasters2 (name, foodLiked, producer) \{foodLiked->producer\}

- Not done! We need to check further violations.
  - FoodTesters1 – No violations!
  - FoodTesters2 – Not BCNF! foodLiked not a key, foodLiked+ no name

- Decompose FoodTesters2 and project dependencies:
  - FoodTasters3 (foodLiked, producer) \{foodLiked->producer\}
  - FoodTasters4 (name, foodLiked)

- Result (satisfies BCNF):
  - FoodTasters1 (name, addr, favFood) \{name->addr, name->favFood\}
  - FoodTasters3 (foodLiked, producer) \{foodLiked->producer\}
  - FoodTasters4 (name, foodLiked)
Some comments on BCNF decomp

◆ If more than one FD violates BCNF, you may decompose based on any one of them.
  ♦ Because of this, there may be multiple possible results.

◆ The new relations we create may not be in BCNF. We must recurse.
  ♦ We only keep the relations at the “leaves”.
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!).
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).
Exercises

- Parts(part, manuf, seller, price)
- FDs = {part->manuf; part,seller->price}
- Exercises
How to determine keys from FDs?

◆ **Prime attribute:** if it’s part of any key

◆ **Example:** \( R(ABC), \text{ FDs} = \{A\rightarrow B, B\rightarrow C\} \)
  - A is clearly a key
  - \( => \) A is prime, B and C are non-prime

◆ **R(ABC), FDs = \{AB\rightarrow C, C\rightarrow A\}**
  - AB and BC are the keys (Check: closure test!)
  - \( => \) A, B, and C are prime (but not keys alone!)

◆ How do I know AB and BC are keys?
How to determine keys from FDs?

- $R(ABC)$, FDs = \{A$\rightarrow$B, B$\rightarrow$C\}

<table>
<thead>
<tr>
<th>Left</th>
<th>Middle</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
</tbody>
</table>

- Only on Left $\Rightarrow$ must be part of any key!
- Only on Right $\Rightarrow$ cannot be part of any key!
- Middle $\Rightarrow$ maybe, maybe not.
- In this case: $A^+ = ABC \Rightarrow A$ is the key
- $A$ is part of any key, and it happens to be a key $\Rightarrow$ no need to look at B
How to determine keys from FDs?

- **R(ABCD)**, **FDs** = \{**AB**->C, **C**->B, **C**->D\}

<table>
<thead>
<tr>
<th>Left</th>
<th>Middle</th>
<th>Right</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B C</td>
<td>D</td>
</tr>
</tbody>
</table>

- **A** might be a key, look at **A**\(^+\) = A
- **Add** one from Middle: **AB**\(^+\) = ABCD
- **Must** try it with the other Middles too: **AC**\(^+\) = ACBD
- => both **AB**, **AC** are keys!
- **Which** ones are prime attributes?
- **What** if all are in the middle?
  - Try one at a time, then combinations of two, etc.
Announcements

◆ Optional Xquery tutorial with extra practice exercises
  ❖ Thursday, Nov 26, 5:10pm-7pm, in MC252, OR
  ❖ Friday, Nov 27, 10:10am-12pm, in GB220
  ❖ Don’t have to go to both, it’s the same thing

◆ Remember to start early on A3!
  ❖ We’ll likely post one more extra helper tutorial for FDs/BCNF.
  ❖ Extra office hours closer to deadline

◆ Course evaluations should be out
  ❖ http://uoft.me/course-evals
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* (no more, no less).

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.
  ◆ Example:

<table>
<thead>
<tr>
<th>VIN#</th>
<th>Brand</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>Honda</td>
<td>Red</td>
</tr>
<tr>
<td>11111</td>
<td>Ford</td>
<td>Blue</td>
</tr>
<tr>
<td>22222</td>
<td>Ford</td>
<td>Black</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VIN#</th>
<th>Brand</th>
<th>Colour</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345</td>
<td>Honda</td>
<td></td>
</tr>
<tr>
<td>11111</td>
<td>Ford</td>
<td></td>
</tr>
<tr>
<td>22222</td>
<td>Ford</td>
<td></td>
</tr>
</tbody>
</table>

◆ [Exercise]

Notice anything about R₁ △ R₂?
What BCNF decomposition offers

1. **No anomalies**: ✓ (Due to no redundancy)
2. **Lossless Join**: ✓ (see next slide for important observation)
3. **Dependency Preservation**: ×
The BCNF property does not guarantee lossless join

- If you use the BCNF decomposition algorithm, a lossless join is guaranteed.
- Not if the schema just happens to be in BCNF
  - If you generated 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 see the basic algorithm for this check more formally later (Chase test).
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.
What do we do if ...  

◆ There is one structure of FDs that causes trouble when we decompose.  
◆ $AB \rightarrow C$ and $C \rightarrow B$.  
   ◦ Example: $A =$ street address, $B =$ city, $C =$ zip code.  
◆ There are two keys, $\{A,B\}$ and $\{A,C\}$.  
◆ $C \rightarrow B$ is a BCNF violation, so we must decompose into $R1 = BC$ (because $C^+ = BC$), and $R2 = AC$ (recall: $R - C^+ + C$).
We Cannot Enforce FDs

- The problem is that if we use \( R_1 = AC \) and \( R_2 = BC \) as our database schema, we cannot enforce the FD \( AB \rightarrow C \) by checking FDs in these decomposed relations.

- Example with \( A = \text{street}, \ B = \text{city}, \ \text{and} \ C = \text{zip} \) on the next slide.
An Unenforceable FD

\[ A = \text{street}, \ B = \text{city}, \ C = \text{zip} \]

Previous FDs: \( AB \rightarrow C, \ C \rightarrow B \)  => a. \( \text{street,city} \rightarrow \text{zip} \)  b. \( \text{zip} \rightarrow \text{city} \)

\[
\begin{array}{|c|c|}
\hline
\text{street} & \text{zip} \\
\hline
545 \text{ Tech Sq.} & 02138 \\
545 \text{ Tech Sq.} & 02139 \\
\hline
\end{array}
\]

\[
\begin{array}{|c|c|}
\hline
\text{city} & \text{zip} \\
\hline
\text{Cambridge} & 02138 \\
\text{Cambridge} & 02139 \\
\hline
\end{array}
\]

Natural Join (Join tuples with equal zip codes):

\[
\begin{array}{|c|c|c|}
\hline
\text{street} & \text{city} & \text{zip} \\
\hline
545 \text{ Tech Sq.} & \text{Cambridge} & 02138 \\
545 \text{ Tech Sq.} & \text{Cambridge} & 02139 \\
\hline
\end{array}
\]

Although no FDs were violated in the decomposed relations (b. is still enforced in R2, right?), the FD \text{street city} \rightarrow \text{zip} is violated by the database as a whole.
3NF lets us avoid this problem

- 3rd Normal Form (3NF) modifies the BCNF condition to be less strict, so that we don’t have to decompose in this problem situation.

- Recall: An attribute is *prime* if it is a member of any key.

- \( X \rightarrow A \) violates 3NF if and only if \( X \) is not a superkey, and also \( A \) is not prime.

- I.e., in 3NF, it’s ok if \( X \) is not a superkey as long as \( A \) is prime.
Example

◆ R(ABC)
◆ FDs = \{AB->C, C->B\}

◆ Keys?
  ♦ AB
  ♦ AC

◆ AB->C  => AB is a superkey anyway
◆ C->B    => C not a superkey, but B is prime!
=> R violates BCNF, but is in 3NF!
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.
Simple Example – 3NF Synthesis

- R(A,B,C,D,E), FDs = \{A->B, CD->E\}

3NF Synthesis:
- Compute all keys for R (recall simpler algorithm last time)
  - Keys: \{ACD\}
- Find the Minimal Cover/Basis: \{A->B, CD->E\}
- Use FDs in Minimal Basis to define new relations:
  - R1(A,B), R2(C,D,E)
- If none are a superkey for R, add relation whose schema is some key
  - Add R3(A, C, D)
- => The decomposition R1, R2, R3 is in 3NF.
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 \textit{not} a superkey, but \( A \) is at least prime.
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 out 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 3\textsuperscript{rd} 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**: It’ll be clearer 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 never need to test for lossless join, IF the schema we have has been generated through the BCNF decomposition or 3NF synthesis algorithms.
◆ But merely satisfying BCNF or 3NF does not guarantee a lossless join!
The Chase Test

An organized way to see if a tuple \( t \) in the natural join of subschemas \( R_i \), to be a tuple in original \( R \), using the FDs.

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 FDs to show that one of these tuples must be \( t \)?
Setup for the Chase Test

- Start by assuming \( t = abc \ldots \).
- For each \( i \), there is a tuple \( s_i \) of \( R \) that has \( a, b, c, \ldots \) 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).
How to check if a join is lossless?

Problem: Given a decomposed relation, check if decomposed tables are able to produce the EXACT original table

Slightly Simpler Algorithm:

I. represent FDs for each relation in a table

II. Try to get at least one row to have all attributes, using these rules, applied repeatedly to FDs:

1. At least 2 rows with attributes for LHS of FD, and
2. At least 1 row with attribute for RHS of FD, and
3. At least 1 row with attribute not in RHS of FD

Example: R(A,B,C,D,E), {A->B, A->C, D->C, BD->E}

R1(A,B,D)

R2(B,C,D,E)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>R2</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

=> Lossless!
Another example

◆ Problem: Given a decomposed relation, check if decomposed tables are able to produce the EXACT original table

◆ Algorithm: I. represent FDs for each relation in a table
  II. Try to get at least one row to have all attributes.
    ♦ 1. At least 2 rows with attributes for LHS of FD, and
    ♦ 2. At least 1 row with attribute for RHS of FD, and
    ♦ 3. At least 1 row with attribute not in RHS of FD

◆ Example:
  ♦ R(A,B,C,D,E),  \{AB->C, B->D, E->A, C->E\}
  ♦ R1(A,C,E)
  ♦ R2(B,C,D)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
<td>x</td>
</tr>
<tr>
<td>R2</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>

=> Lossless!