DESIGN THEORY FOR RELATIONAL DATABASES

csc343, Introduction to Databases
Renée J. Miller and Fatemeh Nargesian and Sina Meraji
Winter 2018
Introduction

• There are always many different schemas for a given set of data.
• E.g., you could combine or divide tables.
• How do you pick a schema? Which is better? What does “better” mean?
• Fortunately, there are some principles to guide us.
Database Design Theory

• It allows us to improve a schema systematically.

• General idea:
  • Express constraints on the relationships between attributes
  • Use these to decompose the relations

• Ultimately, get a schema that is in a “normal form” that guarantees good properties.

• “Normal” in the sense of conforming to a standard.

• The process of converting a schema to a normal form is called **normalization**.
Part I: Functional Dependency Theory
A 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>

- In any domain, there may be relationships between attribute values.
- Perhaps:
  - Every part has 1 manufacturer
  - Every manufacture has 1 address
  - Every seller has 1 address
- If so, this table will have redundant data.
Principle: Avoid redundancy

Redundant data can lead to anomalies.

<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>

- **Update anomaly**: if Hammers ‘R Us moves and we update only one tuple, the data is inconsistent.
- **Deletion anomaly**: If ABC stops selling part 8624 and Lee Valley makes only that one part, we lose track of its address.
Definition of FD

• Suppose R is a relation, and X and Y are subsets of the attributes of R.

• \( X \rightarrow Y \) asserts that:
  
  • If two tuples agree on all the attributes in set \( X \), they must also agree on all the attributes in set \( Y \).

• We say that "\( X \rightarrow Y \) holds in R", or "\( X \) functionally determines \( Y \)."

• An FD constrains what can go in a relation.
More formally...

A → B means:

∀ tuples t₁, t₂,

(t₁[A] = t₂[A]) → (t₁[B] = t₂[B])

Or equivalently:

¬∃ tuples t₁, t₂ such that

(t₁[A] = t₂[A]) ∧ (t₁[B] ≠ t₂[B])
Generalization to multiple attributes

\[ A_1 A_2 \ldots A_m \rightarrow B_1 B_2 \ldots B_n \text{ means:} \]
\[ \forall \text{tuples } t_1, t_2, \]
\[ (t_1[A_1] = t_2[A_1] \land \ldots \land t_1[A_m] = t_2[A_m]) \Rightarrow \]
\[ (t_1[B_1] = t_2[B_1] \land \ldots \land t_1[B_n] = t_2[B_n]) \]

Or equivalently:
\[ \neg \exists \text{tuples } t_1, t_2 \text{ such that} \]
\[ (t_1[A_1] = t_2[A_1] \land \ldots \land t_1[A_m] = t_2[A_m]) \land \]
\[ \neg (t_1[B_1] = t_2[B_1] \land \ldots \land t_1[B_n] = t_2[B_n]) \]
Why “functional dependency”?

- “dependency” because the value of $Y$ depends on the value of $X$
- “functional” because there is a mathematical function that takes a value for $X$ and gives a unique value for $Y$
Equivalent sets of FDs

• When we write a set of FDs, we mean that all of them hold.
• We can very often rewrite sets of FDs in equivalent ways.
• When we say $S_1$ is equivalent to $S_2$ we mean that:
  • $S_1$ holds in a relation iff $S_2$ does.
Splitting rules for FDs

- Can we split the RHS of an FD and get multiple, equivalent FDs?

- Can we split the LHS of an FD and get multiple, equivalent FDs?
Coincidence or FD?

- An FD is an assertion about every instance of the relation.
- You can’t know it holds just by looking at one instance.
- You must use knowledge of the domain to determine whether an FD holds.
FDs are closely related to keys

- Suppose K is a set of attributes for relation R.
- Recall definition of superkey:
  - a set of attributes for which no two rows can have the same values.
- A claim about FDs:
  - K is a superkey for R iff K functionally determines all of R.
FDs are a generalization of keys

• **key:**
  \[ X \rightarrow R \]
  Every attribute

• **Functional dependency:**
  \[ X \rightarrow Y \]
  Not necessarily every attribute

• An FD can be more subtle.
Inferring FDs

• Given a set of FDs, we can often infer further FDs.
• This will be handy when we apply FDs to the problem of database design.
• Big task: given a set of FDs,
  • infer every other FD that must also hold.
• Simpler task: given a set of FDs,
  • check whether a given FD must also hold.
Examples

• If $A \rightarrow B$ and $B \rightarrow C$ hold, must $A \rightarrow C$ hold?

• If $A \rightarrow H$, $C \rightarrow F$, and $F \rightarrow G \rightarrow A \rightarrow D$ hold, must $F \rightarrow A \rightarrow D$ hold? must $C \rightarrow G \rightarrow F \rightarrow H$ hold?

• If $H \rightarrow GD$, $HD \rightarrow CE$, and $BD \rightarrow A$ hold, must $EH \rightarrow C$ hold?

• Aside: we are not generating new FDs, but testing a specific possible one.
Method 1: Prove an FD follows using first principles

• You can prove it by referring back to
  • The FDs that you know hold, and
  • The definition of functional dependency.
• But the Closure Test is easier.
Method 2: Prove an FD follows using the Closure Test

- Assume you know the values of the LHS attributes, and figure out everything else that is determined.
- If it includes the RHS attributes, then you know that LHS $\rightarrow$ RHS
- This is called the closure test.
Y is a set of attributes, S is a set of FDs. Return the closure of Y under S.

\textbf{Attribute\_closure}(Y, S):

\begin{itemize}
\item Initialize $Y^+$ to $Y$
\item Repeat until no more changes occur:
  \begin{itemize}
  \item If there is an FD $LHS \rightarrow RHS$ in S such that $LHS$ is in $Y^+$:
    \begin{itemize}
    \item Add $RHS$ to $Y^+$
    \end{itemize}
  \end{itemize}
\end{itemize}

Return $Y^+$
Visualizing attribute closure

If LHS is in $Y^+$ and $\text{LHS} \rightarrow \text{RHS}$ holds, we can add RHS to $Y^+$.
S is a set of FDs; LHS $\rightarrow$ RHS is a single FD. Return true iff LHS $\rightarrow$ RHS follows from S.

**FD_follows(S, LHS $\rightarrow$ RHS):**

\[
Y^+ = \text{Attribute_closure}(\text{LHS}, S)
\]

return (RHS is in Y+)
Projecting FDs

• Later, we will learn how to **normalize** a schema by decomposing relations. This is the whole point of this theory.

• We will need to know what FDs hold in the new, smaller, relations.

  • We must **project** our FDs onto the attributes of our new relations.
Example

R(A1, ..., An) Set of attributes: A
Decompose into:
- R1(B1, ..., Bk) Set of attributes: B, and
- R2(C1, ..., Cm) Set of attributes: C

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

\[
\begin{align*}
R1 &= \pi_B (R) \\
R2 &= \pi_C (R)
\end{align*}
\]
S is a set of FDs; L is a set of attributes.

Return the projection of S onto L:
all FDs that follow from S and involve only attributes from L.

Project(S, L):
  Initialize T to {}.
  For each subset X of L:
    Compute $X^+$  Close X and see what we get.
    For every attribute A in $X^+$:
      If A is in L:  $X \rightarrow A$ is only relevant if A is in L (we know X is).
      add $X \rightarrow A$ to T.
  Return T.
A few optimizations

• No need to add $X \rightarrow A$ if $A$ is in $X$ itself. It’s a trivial FD.

• These subsets of $X$ won’t yield anything, so no need to compute their closures:
  • the empty set
  • the set of all attributes

• Neither are big savings, but...
An important optimization

• If we find $X^+ = \text{all attributes}$, we can ignore any superset of $X$.
  • It can only give use “weaker” FDs (with more on the LHS).
• This is a big time saver!
Projection is expensive

- Even with these optimizations, projection is still expensive.
- Suppose $R_1$ has $n$ attributes. How many subsets of $R_1$ are there?
Minimal Basis

- We saw earlier that we can very often rewrite sets of FDs in equivalent ways.
- Example: $S_1 = \{A \rightarrow BC\}$ is equivalent to $S_2 = \{A \rightarrow B, A \rightarrow C\}$.
- Given a set of FDs $S$, we may want to find a minimal basis: A set of FDs that is equivalent, but has
  - no redundant FDs, and
  - no FDs with unnecessary attributes on the LHS.
S is a set of FDs. Return a minimal basis for S.

**Minimal_basis(S):**

1. Split the RHS of each FD
2. For each FD $X \rightarrow Y$ where $|X| \geq 2$:
   
   If you can remove an attribute from $X$ and get an FD that follows from $S$:
   
   Do so! (It’s a stronger FD.)

3. For each FD $f$:
   
   If $S - \{f\}$ implies $f$:
   
   Remove $f$ from $S.$
Some comments on minimal basis

• Often there are multiple possible results.
  • Depends on the order in which you consider the possible simplifications.

• After you identify a redundant FD, you must not use it when computing subsequent closures.
... and less intuitive

- When you are computing closures to decide whether the LHS of an FD $X \rightarrow Y$ can be simplified, continue to use that FD.

- You must do (2) and (3) in that order. Otherwise, must repeat until no changes occur.
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
  - $R_1(B_1, \ldots, B_j)$ and $R_2(C_1, \ldots, C_k)$ such that:
    - $R_1 = \pi_{B_1, \ldots, B_j}(R)$
    - $R_2 = \pi_{C_1, \ldots, C_k}(R)$
    - $\{B_1, \ldots, B_j\} \cup \{C_1, \ldots, C_k\} = \{A_1, \ldots, A_n\}$
    - $R_1 \bowtie R_2 = R$
$$R(A_1, \ldots A_n)$$ \hspace{2cm} \text{Set of attributes: } A

Decompose into:

- $$R_1(B_1, \ldots B_j)$$ \hspace{2cm} \text{Set of attributes: } B, \text{ and}
- $$R_2(C_1, \ldots C_k)$$ \hspace{2cm} \text{Set of attributes: } C

$$B \cup C = A,$$ \hspace{2cm} $$R_1 \bowtie_{A} R_2 = R$$

$$R_1 = \pi_B(R)$$ \hspace{2cm} $$R_2 = \pi_C(R)$$

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$$R_1$$</td>
<td>$$\bowtie_{A}$$</td>
</tr>
</tbody>
</table>
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 → 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.
R is a relation; F is a set of FDs.
Return the BCNF decomposition of R, given these FDs.

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]
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:
  - \( r \subseteq r_1 \Join \ldots \Join r_n \)
  - I.e., we will get back every tuple.
- But it may not be the case that:
  - \( r \supseteq r_1 \Join \ldots \Join r \)
  - 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 → 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.

\textbf{[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 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**: 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).
5. [Exercise]