Relational Database Design

- To generate a set of relation schemas that allows
  - to store information without unnecessary redundancy
  - to retrieve desired information easily

- Approach
  - design schema in appropriate normal form

- How to determine whether a schema is in normal form?
  - functional dependency: a collection of constraints
  - a key notion in relational database design

- Undesirable properties of bad design
  1) repetition of information, resulting in wasted space and complicated updates
  2) inability to represent certain information: introduction of null values
  3) loss of information
Database Design

- To avoid undesirable features
  - properties of information repetition and null values suggest decomposition of relation schema
  - property of information loss implies lossy-join decomposition

- Major concern in DB design
  - how to specify constraints on the database and how to obtain lossless-join decomposition that avoids the undesirable properties above
Loss of Information

Borrow = (B-name, Loan#, C-name, Amount)
Amt = \( \Pi_{\text{Amount}, \text{C-name}} (\text{Borrow}) \)
Loan = \( \Pi_{B-name, Loan#, Amount} (\text{Borrow}) \)

\[ \text{Amt} \mid \times \text{Loan} \text{ contains more tuples that the original relation} \]

\[ \rightarrow \quad \Pi_{B-name} (\sigma_{C-name=\text{Jones}} (\text{Amt} \mid \times \text{Loan})) \]
may introduce a wrong answer

- More tuples in Amt \( \mid \times \) Loan implies less information
  \[ \rightarrow \text{lossy-join decomposition} \]
- Reason for such anomaly
  - Amount does not uniquely relate B-name and C-name
  - customers may have loans in the same amount, but not necessarily at the same branch
  - uniqueness is critical: notion of key
- Functional dependency is a generalization of the notion of key (uniqueness)
Functional Dependency

For a relation scheme $R(A_1, \ldots, A_n)$, let $X$ and $Y$ be subsets of attributes $A_1, \ldots, A_n$.

$X$ functionally determines $Y$ ($X \rightarrow Y$) if relation $r(R)$ represents the current instance of the schema $R$, and it is not possible to have two tuples in $r$ that agree in components of all attributes $X$ and disagree on some attributes in $Y$.

- Properties
  - if $X$ is a key, then $X \rightarrow Y$ for any possible set of attributes of $R$
  - FD allows to express constraints that cannot be expressed just using keys

<ex> Lending (B-name, Assets, B-city, Loan#, C-name, Amount)

B-name is not a superkey, since a branch may have many loans to many customers, but we can express the dependency

$B\text{-name} \rightarrow B\text{-city}$
Functional Dependency

<table>
<thead>
<tr>
<th>Assign</th>
<th>Pilot</th>
<th>Flight</th>
<th>Date</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>112</td>
<td>3/11</td>
<td>1325</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>301</td>
<td>3/10</td>
<td>0600</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>105</td>
<td>3/11</td>
<td>1325</td>
<td></td>
</tr>
</tbody>
</table>

- Flight functionally determines Time
  Flight → Time

- For any given Pilot, Date, and Time, there is only one flight
  Pilot, Date, Time → Flight

- Formal definition

  A set of attributes X functionally determines the set Y if
  \( t_1(X) = t_2(X) \) implies \( t_1(Y) = t_2(Y) \)

  or, equivalently
  \( t_1(Y) \neq t_2(Y) \) implies \( t_1(X) \neq t_2(X) \)

  or, equivalently
  for all \( X, Y \subseteq R, |\Pi_Y(\sigma_X=r)| \leq 1 \)
Notes on Functional Dependency

- FD is a statement about the universe as we understand it.
  - FD is a semantic integrity constraints, which must hold for all the tuples in the relation
  - all relations must satisfy all FDs; otherwise they are not correct

- It is not true to say
  "Since $X$ functionally determines $Y$, if we know $X$, we know $Y$."
  Or, "If $X \rightarrow Y$, then $X$ identifies $Y$."

- Let $R$ be a schema, then $X \rightarrow R$ iff $X$ is a superkey of $R$

- Some FDs are trivial
  
  \[ A \rightarrow A \]
  
  \[ X \rightarrow Y \text{ if } Y \subseteq X \]
Use of Functional Dependency

- **Legality test**
  - check whether the relations are legal under a given set of FDs
  - if r is legal under a given set of FDs F, we say r satisfies F.

- **Constraints specification**
  - express constraints on the set of legal relations
  - rules to be used in database design

<table>
<thead>
<tr>
<th>&lt;ex&gt; r</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td>b1</td>
<td>c1</td>
<td>d1</td>
<td></td>
</tr>
<tr>
<td>a1</td>
<td>b2</td>
<td>c1</td>
<td>d2</td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td>b2</td>
<td>c2</td>
<td>d2</td>
<td></td>
</tr>
<tr>
<td>a2</td>
<td>b3</td>
<td>c2</td>
<td>d3</td>
<td></td>
</tr>
<tr>
<td>a3</td>
<td>b3</td>
<td>c2</td>
<td>d4</td>
<td></td>
</tr>
</tbody>
</table>

\[ A \rightarrow C \]
\[ AB \rightarrow D \]
\[ C \rightarrow A \]
Logical Implications of Functional Dependency

- Not enough to consider only the given set of FDs
  - need to consider all FDs that hold
  - for a given set of FDs F, certain other FDs also hold: they are *logically implied* by F

<ex>
R(A, B, C)
FD: {A → B, B → C}
A → B ∧ B → C ⇒ A → C
</ex>

<Proof>
If t₁(A)=t₂(A), then t₁(B)=t₂(B) by A → B.
Since B → C, t₁(C)=t₂(C). Therefore if t₁(A)=t₂(A), then t₁(C)=t₂(C). Hence A → C.
</Proof>

- Closure of F (F⁺)
  - set of all FDs logically implied by F
  - if F = F⁺, F is called a full family of dependencies
  - how to compute F⁺? use inference rules
Inference Rules

- A means of inferring the existence of FD from the given set

- Completeness and soundness
  - completeness: given F, the rules allow us to determine all dependencies in F⁺
  - soundness: we cannot generate any FD not in F⁺

<ex> R=(A, B, C, D) F={A → B, B → C} F⁺ = {A → B, B → C, A → C}

A → D? If we get it by the rules, they are not sound.

If we cannot get A → C by the rules, they are not complete.

- Are there complete and sound inference rules to compute F⁺?

  Armstrong’s axioms
Armstrong’s Axioms

(1) reflexivity rule
   if $Y \subseteq X$, then $X \rightarrow Y$ holds (trivial dependency)

(2) augmentation rule
   if $X \rightarrow Y$, then $XZ \rightarrow YZ$

(3) transitivity rule
   if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

Although these three rules are complete, there are
additional three rules to compute $F^+$ directly

(4) additivity (union) rule
   if $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

(5) projectivity (decomposition) rule
   if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$

(6) pseudo-transitivity rule
   if $X \rightarrow Y$ and $Wy \rightarrow Z$, then $XW \rightarrow Z$
Additional Rules

- Additional rules can be derived from the original rules

union rule:
1. $X \rightarrow Y$ given
2. $X \rightarrow XY$ augment $X$
3. $X \rightarrow Z$ given
4. $XY \rightarrow YZ$ augment $Y$
5. $X \rightarrow YZ$ transitivity 2 & 4

decomposition:
1. $X \rightarrow YZ$ given
2. $YZ \rightarrow Z$ reflexivity
3. $X \rightarrow Z$ transitivity 1 & 2

pseudo-transitivity:
1. $X \rightarrow Y$ given
2. $XW \rightarrow YW$ augment $W$
3. $WY \rightarrow Z$ given
4. $XW \rightarrow Z$ transitivity 2 & 3

- Theorem: Armstrong’s axioms are sound and complete
Computing Closure

Let X be a set of attributes, then $X^+$ is the set of all attributes functionally determined by X under a set of FDs F.

<Algorithm>

Input: F and X
Output: $X^+$ (the closure of X with respect to F)

Compute a sequence of sets of attributes $X^{(0)}$, $X^{(1)}$, ... by the following rule:

1. $X^{(0)} = X$
2. $X^{(i+1)} = X^{(i)} \cup Z$
   if $Y \rightarrow Z \in F$ and $Y \subseteq X^{(i)}$
3. stop when $X^{(i+1)} = X^{(i)}$

<ex> {A → C, B → C, C → D, DE → C, CE → A} 

1. $X = AD$
   $X^{(0)} = \{AD\}$
   $X^{(1)} = \{AD\} \cup \{C\} = \{ACD\}$
   $X^{(2)} = \{ACD\} = X^{(1)}$
   stop
2. $X = BC$
   $X^{(0)} = \{BC\}$
   $X^{(1)} = \{BC\} \cup \{CD\} = \{BCD\}$
   $X^{(2)} = \{BCD\} = X^{(1)}$
   stop
Relation Decomposition

- One of the properties of bad design suggests to decompose a relation into smaller relations
  - must achieve lossless-join decomposition (non-additive join)

<ex> \[ R = (A, B, C) \quad r \quad A \quad B \quad C \]
\[
F = \{ A \rightarrow B \}
\]
\[
\begin{array}{ccc}
\hline
a1 & b1 & c1 \\
a2 & b1 & c2 \\
\hline
\end{array}
\]

Decomposition 1:

\[
\begin{array}{ccc|ccc|ccc}
\hline
r1 & A & B & B & C & r1 \times r2 & A & B & C \\
\hline
a1 & b1 & b1 & c1 & a1 & b1 & c1 & \checkmark \\
a2 & b1 & b1 & c2 & a1 & b1 & c2 & \checkmark \\
\hline
\end{array}
\]

Decomposition 2:

\[
\begin{array}{ccc|ccc|ccc}
\hline
r1 & A & B & A & C & r1 \times r2 & A & B & C \\
\hline
a1 & b1 & a1 & c1 & a1 & b1 & c1 \\
a2 & b1 & a2 & c2 & a2 & b1 & c2 \\
\hline
\end{array}
\]
Lossless Join

If $R$ is a relation schema decomposed into $R_1, ..., R_K$, and $F$ is a set of FDs, the decomposition is called a \textit{lossless join} with respect to $F$, if for every relation $r(R)$ satisfying $F$

$$r = \Pi_{R_1}(r) \times \ldots \times \Pi_{R_K}(r)$$

• $r$ is the natural join of its projection onto $R_i$

• Recoverability

- lossless join property is necessary if the decomposed relation is to be recovered from its decomposition

• Testing lossless join

Let $R$ be a schema and $F$ be a set of FDs on $R$, and $\alpha = (R_1, R_2)$ be a decomposition of $R$. Then $\alpha$ has a lossless join w.r.t. $F$ iff either $R_1 \cap R_2 \rightarrow R_1$ (or $R_1 - R_2$)

or $R_1 \cap R_2 \rightarrow R_2$ (or $R_2 - R_1$)

where such $FD \in F^+$
Lossless Join Decomposition

From the previous example: \( R=(ABC) \quad F=\{ A \rightarrow B \} \)

(1) \( R_1 = (AB) \quad R_2 = (AC) \)
\[ R_1 \cap R_2 = A \quad R_1 - R_2 = B \]
A \(\rightarrow\) B in F? Yes.

(1) \( R_1 = (AB) \quad R_2 = (BC) \)
\[ R_1 \cap R_2 = B \quad R_1 - R_2 = A \quad R_2 - R_1 = C \]
B \(\rightarrow\) A in F? No. \( F^+ \) ? No.
B \(\rightarrow\) C in F? No. \( F^+ \) ? No.

Therefore lossy join

<ex>
\( R=(\text{City}, \text{Street}, \text{Zip}) \quad F=\{ \text{CS} \rightarrow \text{Z}, \text{Z} \rightarrow \text{C} \} \)
\[ R_1 = (CZ) \quad R_2 = (SZ) \]
\[ R_1 \cap R_2 = Z \quad R_1 - R_2 = C \]
Z \(\rightarrow\) C in F? Yes.
Dependency Preservation Decomposition

- Another desirable property of decomposition
  - each FD specified in $F$ either appears directly in one of the relations in the decomposition, or be inferred from FDs that appear in some relation

- Why desirable?
  - when updating the DB, the system must check all the FDs are satisfied
  - for efficiency, violation detection can be done without performing join operation
    \[ \rightarrow \text{FDs need to be tested by checking one relation} \]

- A decomposition preserves a set of FDs $F$, if the union of all FDs in $\Pi_{R_1}(F)$ logically implies all FDs in $F$

  \[ F_i = \Pi_{R_1}(F^+) \]
  \[ F = \bigcup F_i \]
  \[ \text{check if } (F^+)^+ = F^+ \]
Testing Dependency Preservation

<ex> R=(City, Street, Zip) \quad F=\{CS \to Z, Z \to C\}
R_1=(S,Z) \quad R_2=(C,Z)

(1) lossless join?
R_1 \cap R_2 = Z, R_1 - R_2 = C, Z \to C \text{ in } F? \text{ Yes}

(2) dependency preserving?
R_1: \text{ only trivial FD}
R_2: \text{ Z }\to \text{ C and trivial FD}

\( (\Pi_{R_1}(F) )\cup \Pi_{R_2}(F))^+ \neq F^+ \)

They do not imply \( CS \to Z \).
Hence the decomposition does not preserve dependency.

- Algorithm for testing dependency preservation
  - given in the text book, but is not very practical since
  it requires computing \( F^+ \) that takes exponential time
Minimal Redundancy

- Another desirable property of decomposition
  - decomposition should contain as little redundant information as possible
  - degrees to which we can achieve the lack of redundancy is represented by several *normal forms*

- Normalization process
  - first introduced by Codd in 1972
  - a series of tests to certify whether or not a relation schema belongs to a certain normal form
  - Codd proposed three normal forms: 1NF, 2NF, and 3NF
  - stronger 3NF was proposed by Boyce and Codd: BCNF
  - 1NF, 2NF, 3NF, and BCNF are all based on FDs
  - 4NF is based on multivalue dependency
  - 5NF is based on join dependency
  - domain-key normal form represents an ultimate normal form