CS480

Database Systems
6 - Database Design & Normalization
Course webpage
Boris Glavic
bglavic@uic.edu

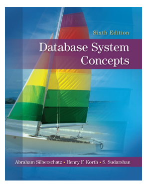
Database Design & Normalization

Relational Database Design

Relational Database Design

- Features of a **Good / Bad Design**
- **Atomic Domains** and **First Normal Form** (**1NF**)
- **Decomposition** as a tool to *"fix"* a bad design (resolve redundancies)
- Identifying bad designs based on **(functional) dependencies** between attributes
  - **Functional dependency theory** and tool box
- **Normal forms** (disallowing redundancies)
  - **1NF**, **2NF**, **3NF**, **Boyce-Codd NF**

Textbook



Textbook: Chapter 7

Bad Design - Redundancy

- Suppose we combine *instructor* and *department* into *inst_dept*
- We saw before that this leads to redundancy (repeated information)

Why Redundancy is Bad

- **Update Physics department**
  - need to update multiple tuples
  - inefficient and potential for errors (if only some copies are updated)
- **Delete Physics department**
  - need to update multiple tuples
  - inefficient and potential for errors (if only some copies are updated)
- **Departments without instructors or instructors without a department**
  - Need dummy department and instructor
  - Makes aggregation error-prone (dummies should not be counted)

Not All Combined Schemas are Bad!

- Combining relations does not always lead to redundancy!

- **secclass** (sec_id, building, room_number)
- **secinfo** (course_id, sec_id, semester, year)
- combined relation: **section** (course_id, sec_id, semester, year, building, room_number)

section

secclass **secinfo**

What Leads to Redundancy?

What does redundancy mean?

- The values of some attributes of a relation are uniquely determined by the values of other attributes

- **instructor** (id, salary, deptname, building, budget)
- `deptname` determines the values of `building` and `budget`

What Leads to Redundancy?

What about keys?

- Note that the above description sounds suspiciously like the definition of a key!
- But keys are needed to identify tuples and are in general unavoidable!
- The issue stems from **attributes that are not a key determining other attributes that are not part of a key**
  - `deptname` is not part of the key so
    - there may be multiple tuples with the same department
    - these tuples will all have the same `building` and `budget`

Functional dependencies

- We need some generalization of keys to express that some attributes determine some, but not all other attributes of a relation: functional dependencies

Fixing Redundancy - Decomposition

- Decomposition splits a relation into multiple relations *R* by projecting on subsets of the attributes of *R*
  - Each resulting table is called a fragment
- Decomposition can **resolve redundancy**

$\pi_{id,salary,deptname}(instdept)$ $\pi_{deptname,building,budget}(instdept)$

Lossy Decompositions

- Decompositions can loose information, they may be lossy

$\pi_{id,salary}(instdept)$ $\pi_{deptname,building,budget}(instdept)$

$inst \bowtie dept$

Testing For Lossless-ness

- How can we **test whether a decomposition is lossless**?
- If we can **reconstruct** the original table from the decomposed fragments, the apparently we have not lost information!
  - Start with the original table, decompose it, join back the fragments
  - If the result is the same as the original table, then the decomposition is lossless

Remark

- This needs to work for **every valid instance** though!
- Need to determine this based on **integrity constraints** alone
- **Functional dependencies** will allow us to formalize this

Goal - Devise A Theory of Normalization

- Decide whether a particular relation *R* is in ***"good"* form**.
- In the case that a relation *R* is not in *"good"* form, **decompose** it into a set of relations $\{R_1, R_2, \ldots, R_n\}$ such that
  - each relation is in **good form**
  - the decomposition is a **lossless decomposition**
- Our theory is based on:
  1. Models of dependency between attribute values to determine whether decompositions are lossless
     - **functional dependencies**
     - **multivalued dependencies**
  2. Concept of **lossless decomposition**
  3. **Normal Forms** Based On
     - Atomicity of values
     - Avoidance of redundancy
     - Transformation into normal forms by lossless decomposition

Functional Dependency Theory

Relational Database Design

Functional Dependency Theory
- Functional Dependencies
- Inference & Closures
- Armstrong's Axioms
- Attribute Closures
- Canonical Cover

Decomposition & Dependency Preservation

Agenda

- Theory of dependencies
- **Lossless decompositions**
  - Define lossless decompositions
  - Check whether a decomposition will be lossless using dependency theory
- **Normalforms & decomposition**
  - Define normal forms that avoid redundancies
  - Devise algorithms for checking whether a schema is a normal form
  - Devise algorithms to transform schema into a normal form using decomposition

Functional Dependency Theory

Functional Dependency Theory
- Functional Dependencies
- Inference & Closures
- Armstrong's Axioms
- Attribute Closures
- Canonical Cover

Integrity Constraints

- Recall that an integrity constraint σ is a **logical condition** evaluated over a relational database instance *D*
  - If $D \models \sigma$ then *D* is said to **fulfill** the constraint
- If an integrity constraint σ is defined on a relational schema **D**, then only instances *D* that fulfill the constraint are **valid** instances of the schema
  - Integrity constraints restrict the valid instances of a schema
- Integrity constraints we have seen so far:
  - **Keys** (super keys, candidate keys)
  - **Foreign keys**

Functional Dependencies

- A functional dependency (FD) $\alpha \rightarrow \beta$ checks whether for all tuples of a relation the values of a set of attributes α uniquely determine the values of attributes β
- Functional dependencies are a **generalization of keys**
  - Thus, every key is a functional dependency

Functional Dependencies

Definition (Fulfilling FDs)

Given a relational schema **R**, a **functional dependency** $\alpha \rightarrow \beta$ where $\alpha \subseteq$ **R** and $\beta \subseteq$ **R** **holds** on an instance *R* of **R** iff:

$$\forall t_1, t_2 \in R : t_1[\alpha] = t_2[\alpha] \rightarrow t_1[\beta] = t_2[\beta]$$

A	B
1	4
1	5
3	7

- $A \rightarrow B$ does **not** hold
- $B \rightarrow A$ does hold

Functional Dependencies and Keys

- K is a **superkey** for **R** iff $K \rightarrow \mathbf{R}$
- K is a candidate key for **R** iff
  - $K \rightarrow \mathbf{R}$
  - $\nexists \alpha \subset K : \alpha \rightarrow \mathbf{R}$

- not all FDs are superkeys
- **inst_dept** (ID, name, salary, deptname, building, budget)
- We may expect these FDs to hold:

$$deptname \rightarrow building$$
$$ID \rightarrow building$$

Using Functional Dependencies

- **Test whether a relation is valid for a schema with FDs as integrity constraints**
  - We say that R satifies Σ
- **Test whether a join decomposition is lossless** (later)
- **Specify in a schema what relations are valid**
  - We say that Σ hold on R

Warning

- If a specific instance R may satisfy an FD σ that does not mean that the FD holds on all instances of **R**
  - e.g., $name \rightarrow ID$ may hold for one instance of the instructor relation

Trivial Functional Dependencies

Definition (Trivial FD)

An FD σ is trivial if it holds on every possible instance of **R**

Proposition (Subset Condition for Triviality)

- *An FD* $\sigma : \alpha \rightarrow \beta$ *is trivial iff* $\beta \subseteq \alpha$

How To Determine FDs For a Schema?

- As FDs have to hold on all instances of a relation, we can in principle not determine them from a single instance
- There are approaches that automate the discovery of FDs, but these are beyond the scope of this class
- For the purpose of this course, we will assume that FDs have been developed by a domain expert that can determine which constraints would be valid for the domain of interest we are designing a database schema for

Functional Dependency Theory

Implication of Dependencies

Definition (Implication)

Consider a set of functional dependencies Σ and single FD σ over the same schema **R**. We say that Σ implies σ written as $\Sigma \Rightarrow \sigma$ iff:

$$\forall D : D \models \Sigma \rightarrow D \models \sigma$$

We can extend this to sets of FDs as follows:

$$\Sigma_1 \Rightarrow \Sigma_2 \Leftrightarrow \forall \sigma \in \Sigma_2 : \Sigma_1 \Rightarrow \sigma$$

Implication Example

$\{A \rightarrow B, B \rightarrow C\}$ implies $A \rightarrow C$

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a2	b2	c2	d3
a2	b2	c2	d4
a2	b2	c2	d5

Equivalence

Definition (Equivalence)

Two sets of FDs Σ_1 and Σ_2 are equivalent ($\Sigma_1 \equiv \Sigma_2$) if they imply each other (they hold on exactly the same set of databases):

$$\Sigma_1 \equiv \Sigma_2 \Leftrightarrow (\Sigma_1 \Rightarrow \Sigma_2) \wedge (\Sigma_2 \Rightarrow \Sigma_1)$$

Closure

Definition (Closure)

The closure Σ^+ of a set of FDs Σ is the set of all FDs implied by Σ:

$$\Sigma^+ = \{\sigma \mid \Sigma \Rightarrow \sigma\}$$

Question

Can this be checked by looking only at the FDs or do we need to look at all infinitely many possible databases?

Theorem (Uniqueness)

If $\Sigma_1 \equiv \Sigma_2$ *then* ${\Sigma_1}^+ = {\Sigma_2}^+$

Properties of The Closure

- Note that the closure of Σ is exponential in the number of the attributes of **R**
  - e.g., there are already an **exponential** number of **trivial** FDs
- The closure of Σ is always a superset of Σ (every FD trivially implies itself)

Functional Dependency Theory

Armstrong's Axioms

Definition (Amstrong's Axioms)

Consider a schema **R**, Armstrong's axioms are

- Reflexivity:
  - Given $\beta \subseteq \alpha \subseteq \mathbf{R}$
  - then $\alpha \rightarrow \beta$
- Augmentation:
  - Given $\sigma_1 : \alpha \rightarrow \beta$ and $\gamma \subseteq \mathbf{R}$
  - then $\alpha \cup \gamma \rightarrow \beta \cup \gamma$
- Transitivity
  - Given $\sigma_1 : \alpha \rightarrow \beta$ and $\sigma_2 : \beta \rightarrow \gamma$
  - then $\alpha \rightarrow \gamma$

Inference with Armstrong's Axioms

Inference

Given a set of Σ,

- we write $\Sigma \rightarrow_{\mathbb{A}} \sigma$ to denote that σ can be derived from Σ by a single application of one of Armstrong's axioms.
- we write $\Sigma \xrightarrow{*}_{\mathbb{A}} \sigma$ to denote that σ can be derived from Σ through some sequence of applications of Armstrong's axioms

We are also interested in the set of all FDs $\Sigma_{\mathbb{A}}$ that can be derived from Σ using Armstrong's axioms:

$$\Sigma_{\mathbb{A}} = \{\sigma \mid \Sigma \xrightarrow{*}_{\mathbb{A}} \sigma\}$$

Soundness and Completeness

A set of inference rules is ...

- sound if all FDs derived by the rules are implied by Σ
- complete if all FDs in $\sigma \in \Sigma^+$ can be inferred using the rules

Theorem (Amstrong's Axioms are Sound and Complete)

Armstrong's axioms are sound and complete:

$$\Sigma_{\mathbb{A}} = \Sigma^+$$

Applying Armstrong's Axioms

- $\mathbf{R} = (A, B, C, G, H, I)$

$$\Sigma = \{ A \rightarrow B,\ A \rightarrow C,\ C, G \rightarrow H,\ C, G \rightarrow I,\ B \rightarrow H \}$$

- **some members of Σ^+**
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ and $B \rightarrow H$
  - $A, G \rightarrow I$
    - augmenting $A \rightarrow C$ with G to get $A, G \rightarrow C, G$
    - transitivity with $C, G \rightarrow I$
  - $C, G \rightarrow H, I$
    - augment $C, G \rightarrow I$ to get $C, G \rightarrow C, G, I$
    - augment $C, G \rightarrow H$ to get $C, G, I \rightarrow H, I$
    - transitivity

Deriving Additional Inference Rules

- Based on the result from the previous slide Armstrong's axioms are sufficient for computing Σ^+
- Prove additional rules that simplify the process (less inference steps)

Prove or disprove the following rules

- $A \rightarrow B, C$ **implies** $A \rightarrow B$ And $A \rightarrow C$
- $A \rightarrow B$ and $A \rightarrow C$ **implies** $A \rightarrow B, C$
- $A, B \rightarrow B, C$ **implies** $A \rightarrow C$
- $A \rightarrow B$ and $C \rightarrow D$ **implies** $A, C \rightarrow B, D$

Deriving Additional Inference Rules - Results

- $A \rightarrow B, C$ **implies** $A \rightarrow B$ And $A \rightarrow C$ (decomposition)
  - $B, C \rightarrow B$ (reflexivity)
  - $A \rightarrow B$ (transitivity)
  - symmetric proof for $A \rightarrow C$
- $A \rightarrow B$ and $A \rightarrow C$ **implies** $A \rightarrow B, C$ (union)
  - $A \rightarrow A, B$ (augment $A \rightarrow B$ with A)
  - $A, B \rightarrow B, C$ (augment $A \rightarrow C$ with B)
  - $A \rightarrow B, C$ (transitivity)
- $A, B \rightarrow B, C$ **implies** $A \rightarrow C$ (wrong), counterexample:

A	B	C
a1	b1	c1
a1	b2	c2

Deriving Additional Inference Rules - Results

- $A \rightarrow B$ and $C \rightarrow D$ **implies** $A, C \rightarrow B, D$ (composition)
  - $A, C \rightarrow B, C$ (augment $A \rightarrow B$ with C)
  - $B, C \rightarrow B, D$ (augment $C \rightarrow D$ with B)
  - $A, C \rightarrow B, D$ (transitivity)

Computing Closures

- We can use the following fix point process

Algorithm 1: Compute FD Closure

Input : Set of FDs Σ, Schema **R**
Output: The closure Σ^+

$\Sigma_{cur} = \emptyset, \Sigma_{new} = \Sigma$
while $\Sigma_{cur} \neq \Sigma_{new}$ **do** `/* until a fix point is reached */`
    $\Sigma_{cur} \leftarrow \Sigma_{new}$
    for $\alpha \subseteq \beta \subseteq \mathbf{R}$ **do** `/* reflexivity */`
        $\Sigma_{new} \leftarrow \Sigma_{new} \cup \{\alpha \rightarrow \beta\}$
    for $\alpha \rightarrow \beta \in \Sigma_{cur}, \gamma \subseteq \mathbf{R}$ **do** `/* augmentation */`
        $\Sigma_{new} \leftarrow \Sigma_{new} \cup \{\alpha \cup \gamma \rightarrow \beta \cup \gamma\}$
    for $\alpha \rightarrow \beta \in \Sigma_{cur} \wedge \beta \rightarrow \gamma \in \Sigma_{cur}$ **do** `/* transitivity */`
        $\Sigma_{new} \leftarrow \Sigma_{new} \cup \{\alpha \rightarrow \gamma\}$
return Σ_{new}

Computing Closures Computational Complexity

Exponential Complexity

- There are obvious ways to improve the algorithm such as computing trivial FDs upfront
- However, the problem is the **exponential output size**
  - no matter what great algorithm we come up with it has to enumerate exponentially many results!

Functional Dependency Theory

Attribute Closure

Definition (Attribute Closure)

Given Σ over **R** and $\alpha \subseteq$ **R**, the attribute closure α^+ of α wrt. Σ is the maximal subset of **R** implied by α:

- $\Sigma \Rightarrow \alpha \rightarrow \alpha^+$
- $\nexists \gamma \supset \alpha^+ : \Sigma \Rightarrow \alpha \rightarrow \gamma$

Computing Attribute Closures

Algorithm 2: Compute Attribute Closure

Input : Set of FDs Σ, Attributes $\alpha \subseteq$ **R**
Output: The attribute closure α^+

```
α_cur = ∅, α_new = α
while α_cur ≠ α_new do                          /* until a fix point is reached */
    α_cur ← α_new
    for β → γ ∈ Σ do
        if β ⊆ α_new then                       /* LHS is in α_new then add RHS */
            α_new ← α_new ∪ γ
return α_new
```


Attribute Closures Computational Complexity

- Let $n = |$ **R** $|$ and $m = | \Sigma |$
- Each each iteration of the outer loop we either add another attribute or stop
  - $\Rightarrow$ we will do at most n iterations of the outer loop
- The inner loop always iterates exactly m times
- $\Rightarrow$ the algorithm is $O(n \cdot m)$
  - much faster than the closure algorithm!

Use Cases of Attribute Closure

- **Testing for a superkey**
  - If $\alpha^+ =$ **R** then α is a super key
- **Testing functional dependencies**
  - If $\beta \subseteq \alpha^+$ then $\Sigma \Rightarrow \alpha \rightarrow \beta$
- **Computing closures** (still exponential so we will not use this)
  - For each $\alpha \subseteq$ **R** compute α^+ and for each subset $\beta \subseteq \alpha^+$ output $\alpha \rightarrow \beta$

Linear Time Attribute Closure Algorithm

- The attribute closure algorithm has **two sources of inefficiency**:
  - Functional dependencies that have *"fired"* in a previous iteration are tested again in all following iterations
  - No progress is monitored for *"finding"* attributes from the LHS of an FD
- The algorithm presented on the next slide from [1] addresses these shortcomings by tracking which attributes from the LHS of an FD have been found so far and which FDs' RHS have been added to the result so far

Linear Time Attribute Closure Algorithm - Data Structures

- **Data Structures**
  - Assign numeric identifiers to the FDs and attributes (starting from 0).
  - `int[] c`: an integer array with one element per FD that is initialized to the size of the LHS of the FD.
  - `list<int>[] rhs`: an array of lists with one element per FD. For each FD stores the numeric IDs of attributes from the FDs RHS.
  - `list<int>[] lhs`: an array of lists of integers, one element per attribute. The element for each attribute stores the IDs of the FDs that have this attribute in its LHS.
  - `set<int> aplus`: a set storing the attributes that we have determined to be in the result so far
  - `stack<int> todo`: a stack of attributes to be processed

Linear Time Attribute Closure Algorithm

Algorithm 3: Compute Attribute Closure (linear time)

Input : Set of FDs Σ, Attributes $\alpha \subseteq$ **R**
Output: The attribute closure α^+

```
todo = α, aplus = ∅
while ¬ todo.isEmpty() do                       /* until todo is empty */
    curA ← todo.pop()
    aplus.add(curA)                             /* add curA to result */
    for fd ∈ lhs[curA] do                       /* update LHS attributes found so far */
        c[fd] − −                               /* found a LHS attr for fd */
        if c[fd] = 0 then
            remove(lhs[curA], fd)               /* avoid firing twice */
            for newA ∈ rhs[fd] do               /* add implied attributes */
                if ¬ aplus[newA] then           /* if attribute is new add to todo */
                    todo.push(newA)
                aplus.add(newA)
return aplus
```


Functional Dependency Theory

Motivation

- Sets of FDs may contain redundant dependencies that can be inferred from the remaining FDs

- $A \rightarrow C$ is redundant (transitivity) in $\{A \rightarrow B; B \rightarrow C; A \rightarrow C\}$

- Some FDs may have attributes that can be removed without changing the semantics of the set of FDs

- $\{A \rightarrow B; B \rightarrow C; A \rightarrow C, D\}$ can simplified to $\{A \rightarrow B; B \rightarrow C; A \rightarrow D\}$
- $\{A \rightarrow B; B \rightarrow C; A, C \rightarrow D\}$ can simplified to $\{A \rightarrow B; B \rightarrow C; A \rightarrow D\}$

Extraneous Attributes

Definition (Extraneous Attributes)

- Consider a set of FDs Σ and $\sigma : \alpha \rightarrow \beta \in \Sigma$
  - Attribute $A \in \alpha$ is extraneous in α if
    - $\Sigma \Rightarrow (\Sigma - \{\sigma\}) \cup \{(\alpha - \{A\}) \rightarrow \beta\}$
  - Attribute $A \in \beta$ is extraneous in β if
    - $(\Sigma - \{\sigma\}) \cup \{\alpha \rightarrow (\beta - \{A\})\} \Rightarrow \Sigma$
- Technically we require logical equivalence, but the other direction is trivial as *"stronger"* FDs always imply *"weaker"* ones

Extraneous Attributes Example

- $\Sigma = \{A \rightarrow C; A, B \rightarrow C\}$
  - B is extraneous in $A, B \rightarrow C$ because Σ implies $A \rightarrow C$
- $\Sigma = \{A \rightarrow C; A, B \rightarrow C, D\}$
  - C is extraneous in $A, B \rightarrow C, D$ since $A, B \rightarrow C$ can be inferred even after deleting C

Testing for Extraneous Attributes

- Consider $\sigma : \alpha \rightarrow \beta$ such that $\sigma \in \Sigma$
- **Testing if $A \in \alpha$ is extraneous in α**
  - compute $(\alpha - \{A\})^+$ using Σ
  - if $\beta \subseteq (\alpha - \{A\})^+$ then A is extraneous in α
- **Testing if $A \in \beta$ is extraneous in β**
  - compute α^+ using $\Sigma' = (\Sigma - \{\sigma\}) \cup \{\alpha \rightarrow (\beta - \{A\})\}$
  - if α^+ contains A then A is extraneous in β

Canonical Cover

Definition (Canonical Cover)

A set of FDs Σ_C is a canonical cover of a set of FDs Σ iff:

- $\Sigma \equiv \Sigma_C$
- No FD in Σ_C contains an extraneous attribute
- No two FDs in Σ_C share the same LHS

Computing Canonical Covers

Algorithm 4: Compute Canonical Cover

Input : Set of FDs Σ
Output: A Canonical Cover Σ_C

```
Σcur = ∅, Σnew = Σ
while Σcur ≠ Σnew do                                  /* until a fix point is reached */
    Σcur ← Σnew
    for σ1 : α → β1, σ2 : α → β2 ∈ Σ do                /* union RHS */
        Σnew ← Σnew − {σ1, σ2} ∪ {α → β1 ∪ β2}
    for σ : α → β ∈ Σ do
        if A ∈ α is extraneous then
            Σnew ← Σnew − {σ} ∪ {(α − {A}) → β}
            continue
        if A ∈ β is extraneous then
            Σnew ← Σnew − {σ} ∪ {α → (β − {A})}
            continue
return Σnew
```


Computing Canonical Covers

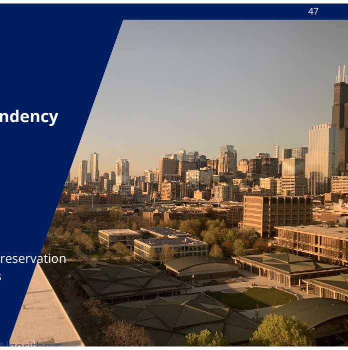
$\mathbf{R} = (A, B, C)$

$\Sigma = \{$
$A \rightarrow B, C$
$B \rightarrow C$
$A \rightarrow B$
$A, B \rightarrow C$
$\}$

- **Union**: Combine $A \rightarrow B, C$ and $A \rightarrow B$ into $A \rightarrow B, C$
  - *Intermediate result* $\{A \rightarrow B, C; B \rightarrow C; A, B \rightarrow C\}$
- **Removing extraneous attributes**: A is extraneous in $A, B \rightarrow C$
  - Check if after deleting A the FD is implied by Σ
    - yes, $B \rightarrow C$ is in the set
  - *Intermediate result* $\{A \rightarrow B, C; B \rightarrow C\}$
- **Removing extraneous attributes**: C is extraneous in $A \rightarrow B, C$
  - Check if $A \rightarrow C$ is implied by $A \rightarrow B$ and the other dependencies
    - yes, using transitivity on $A \rightarrow B$ and $B \rightarrow C$
- **The canonical cover is:**

$$\Sigma_C = \{A \rightarrow B; B \rightarrow C\}$$

Decomposition & Dependency Preservation



Agenda

- **Theory of dependencies**
- Lossless decompositions
  - Define lossless decompositions
  - Check whether a decomposition will be lossless using dependency theory
- **Normalforms & decomposition**
  - Define normal forms that avoid redundancies
  - Devise algorithms for checking whether a schema is a normal form
  - Devise algorithms to transform schema into a normal form using decomposition

Decomposition & Dependency Preservation

Lossless Join Decomposition

Definition (Decomposition)

Given a relational schema $\mathbf{R}(A_1, \ldots, A_n)$ and an instance R over $\mathbf{R}$ and sets of attributes $\mathbf{R_1}, \ldots, \mathbf{R_m}$ such that $\forall i \in [1, m] : \mathbf{R_i} \subseteq \mathbf{R}$ is called a decomposition of $\mathbf{R}$.
The decomposition of R wrt. $\mathbf{R_1}, \ldots, \mathbf{R_m}$ is this set of instances:

$$\{R_i \mid R_i = \pi_{\mathbf{R_i}}(R)\}$$

Lossless Join Decomposition

Definition (Lossless Join Decomposition)

Consider a decomposition $\mathbf{R_1}, \ldots, \mathbf{R_m}$ of a schema $\mathbf{R}(A_1, \ldots, A_n)$. We call $\mathbf{R_1}, \ldots, \mathbf{R_m}$ a lossless join decomposition of **R** if for **every** instance R of **R** we have:

$$R = \pi_{\mathbf{R_1}}(R) \bowtie \ldots \bowtie \pi_{\mathbf{R_m}}(R)$$

Decomposition & Dependency Preservation

Sufficient Condition for Lossless Join Decomposition

- How can we test whether a decomposition will be lossless?

Theorem (Sufficient Condition)

Consider schema **R** *with functional dependencies* Σ*. A decomposition* $\mathbf{R_1}$ *and* $\mathbf{R_2}$ *is lossless if at least one of the following FDs is in* Σ^+*:*

- $\mathbf{R_1} \cap \mathbf{R_2} \rightarrow \mathbf{R_1}$
- $\mathbf{R_1} \cap \mathbf{R_2} \rightarrow \mathbf{R_2}$

How does This Condition Work?

Why does this work?

- WLOG let us assume that $\mathbf{R_1} \cap \mathbf{R_2} \rightarrow \mathbf{R_2}$ holds
- If the common attributes determine all attributes of $\mathbf{R_2}$, then $A = \mathbf{R_1} \cap \mathbf{R_2}$ is a key for $\mathbf{R_2}$
- Consider a tuple $t \in R_1$. Then the values of $t.A$ determine all the values of a tuple in R_2
  - — $\Rightarrow$ each tuple $t \in R_1$ will join with **exactly one** tuple in R_2
  - — $\Rightarrow$ Consider a tuple $t \in R$ that was decomposed into $t_1 \in R_1$ and $t_2 \in R_2$. The natural join of $R_1 \bowtie R_2$ will reconstruct t

The Sufficient Condition in Action

- $\mathbf{R} = (A, B, C)$ with $\Sigma = \{A \rightarrow B; B \rightarrow C\}$
- **Decomposition** $\mathbf{R_1} = (A, B)$ and $\mathbf{R_2} = (B, C)$
  - — this is a **lossless join decomposition**
  - — $\mathbf{R_1} \cap \mathbf{R_2} = \{B\}$ and $B \rightarrow B, C \in \Sigma^+$

R

A	B	C
1	1	1
2	1	1
3	2	3
4	2	3

R_1

A	B
1	1
2	1
2	2
4	2

R_2

B	C
1	1
2	3

$R_1 \bowtie R_2$

A	B	C
1	1	1
2	1	1
3	2	3
4	2	3

The Sufficient Condition in Action

- $\mathbf{R} = (A, B, C)$ with $\Sigma = \{A \rightarrow B; C \rightarrow B\}$
- **Decomposition** $\mathbf{R_1} = (A, B)$ and $\mathbf{R_2} = (B, C)$
  - — this is **not a lossless join decomposition**
  - — $\mathbf{R_1} \cap \mathbf{R_2} = \{B\}$
  - — $B \rightarrow B, C \notin \Sigma^+$
  - — and $B \rightarrow A, B \notin \Sigma^+$

R

A	B	C
1	1	1
2	1	3

R_1

A	B
1	1
2	1

R_2

B	C
1	1
1	3

$R_1 \bowtie R_2$

A	B	C
1	1	1
2	1	1
1	1	3
2	1	3

Decomposition & Dependency Preservation

Dependencies on Decomposed Relations

- What happens to dependencies under decompositions?
- We can only directly check dependencies $\alpha \rightarrow \beta$ where $\alpha \cup \beta$ is contained in at least one fragment $\mathbf{R_i}$

Definition (Dependency Preservation)

For a decomposition $\mathbf{R_1}, \ldots, \mathbf{R_n}$ of **R** with FDs Σ we define:

$$\Sigma_i = \{\alpha \rightarrow \beta \mid \alpha \rightarrow \beta \in \Sigma^+ \wedge (\alpha \cup \beta) \subseteq \mathbf{R_i}\}$$

The decomposition is dependency preserving if:

$$\left(\bigcup_i \Sigma_i\right)^+ = \Sigma^+$$

Dependency Preservation

Caveat

- note that Σ_i is defined using the closure Σ^+ and, thus, may be exponentially large!

Why do we need the closure?

- $\Sigma = \{A \rightarrow B; B \rightarrow C\}$ over $\mathbf{R} = (A, B, C)$
- Consider decomposition $\mathbf{R_1} = (AC)$ and $\mathbf{R_2} = (AB)$
- Σ_1 includes $A \rightarrow C$ as $A \rightarrow C$ is in Σ^+ and only uses attributes from $\mathbf{R_1}$
- However, $A \rightarrow C$ is not present in Σ

Testing Dependency Preservation - Naive Algorithm

Algorithm 5: Test Dependency Preservation (naive)

Input : Set of FDs Σ, Decomposition $\mathbf{R_1}, \ldots \mathbf{R_n}$

Output: **True** if the decomposition preserves Σ

for $i \in [1, n]$ **do**
  $\Sigma_i = \{\alpha \rightarrow \beta \mid \alpha \rightarrow \beta \in \Sigma^+ \wedge (\alpha \cup \beta) \subseteq \mathbf{R}_i\}$
$\Sigma_{decomposed} = \bigcup_{i=1}^{n} \Sigma_i$
return ${\Sigma_{decomposed}}^+ = \Sigma^+$

Testing Dependency Preservation

- Apply the PTIME procedure shown on the next slide to each $\sigma \in \Sigma$.
  - If it returns **true** for each $\sigma \in \Sigma$, then the **decomposition is dependency preserving**.
  - If it fails however, we have to fall back to the test using closures
- That is: returning **true** for all $\sigma \in \Sigma$ is a **sufficient**, but not **necessary** condition for dependency preservation

Sufficient Test for Dependency Preservation

Algorithm 6: Test Dependency Preservation

Input : Set of FDs Σ and $\sigma : \alpha \rightarrow \beta \in \Sigma$, Decomposition $\mathbf{R_1}, \ldots \mathbf{R_n}$
Output: True if the decomposition preserves σ

```
A_cur ← ∅
A_new ← α
while A_cur ≠ A_new do                          /* until a fix point is reached */
    A_cur ← A_new
    for i ∈ [1, n] do
        A_add ← (A_new ∩ R_i)^+ ∩ R_i
        A_new ← A_new ∪ A_add
return β ∈ A_new
```


Why Does The Sufficient Condition Work

1. $\alpha \rightarrow \beta \in \Sigma$ is preserved in the decomposition if $\alpha^+ \supseteq \beta$ when α^+ is computed using $\Sigma_{decomposition} = \bigcup_{i=1}^{n} \Sigma_i$
   - the decomposition is dependency preserving if and only if all $\sigma \in \Sigma$ are preserved (as then we can infer any $\sigma \in \Sigma^+$ using $\Sigma_{decomposition}$)
2. We still need to show that if the algorithm returns **true**, then $\alpha \rightarrow \beta \in \Sigma$ is preserved under the decomposition
   - for any $\gamma \subseteq \mathbf{R}_i$, $\gamma \rightarrow \gamma^+$ is an FD in Σ^+ (follows from the definition of attribute closure)
   - then $\gamma \rightarrow \gamma^+ \cap \mathbf{R}_i$ will be an FD in ${\Sigma_{decomposition}}^+$ (based on the definition of $\Sigma_{decomposition}$
   - for any FD $\gamma \rightarrow \delta$ is in $\Sigma_i \subseteq \Sigma_{decomposition}$ if $\delta \subseteq \gamma^+ \cap \mathbf{R}_i$

Positive Example

- $\mathbf{R} = (A, B, C)$ with $\Sigma = \{A \rightarrow B, B \rightarrow C\}$
- **Decomposition** $\mathbf{R_1} = (A, B)$ and $\mathbf{R_2} = (B, C)$
  - this **lossless join decomposition** is **dependency preserving**

R

A	B	C
1	1	1
2	1	1
3	2	3
4	2	3

R_1

A	B
1	1
2	1
3	2
4	2

R_2

B	C
1	1
2	3

$R_1 \bowtie R_2$

A	B	C
1	1	1
2	1	1
3	2	3
4	2	3

Negative Example

- $\mathbf{R} = (A, B, C)$ with $\Sigma = \{A \rightarrow B, B \rightarrow C\}$
- **Decomposition** $\mathbf{R_1} = (A, B)$ and $\mathbf{R_2} = (A, C)$
  - this is a **lossless join decomposition**
  - not dependency preserving ($B \rightarrow C$ is not preserved)

R

A	B	C
1	1	1
2	1	1
3	2	3
4	2	3

R_1

A	B
1	1
2	1
2	2
4	2

R_2

A	C
1	1
2	1
3	3
4	3

$R_1 \bowtie R_2$

A	B	C
1	1	1
2	1	1
3	2	3
4	2	3

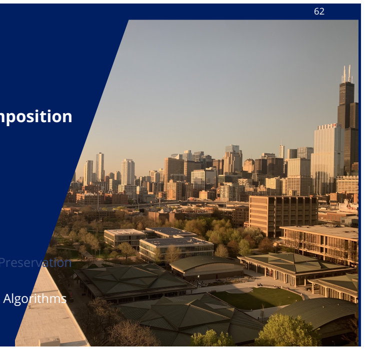
Normalforms & Decomposition Algorithms

Relational Database Design

Functional Dependency Theory

Decomposition & Dependency Preservation

Normalforms & Decomposition Algorithms
- Normal Forms
- 1NF



Agenda

- **Theory of dependencies**
- **Lossless decompositions**
  - Define lossless decompositions
  - Check whether a decomposition will be lossless using dependency theory
- Normalforms & decomposition
  - Define normal forms that avoid redundancies
  - Devise algorithms for checking whether a schema is a normal form
  - Devise algorithms to transform schema into a normal form (**normalize** it) using **decomposition**

Purpose of Normalization

- Consider relation **R** with FDs Σ
- Determine whether **R** is prevents redundancy
- If **R** does allow for certain types of redundancy then decompose it
  - Each fragment is in the desired normal form
  - The decomposition is lossless
  - If possible, the decomposition should be dependency preserving

Normalforms & Decomposition Algorithms

Normalforms & Decomposition Algorithms
- Normal Forms
- 1NF
- 2NF
- 3NF
- BCNF

Outline

- We will cover several normal forms that are increasingly strict, but also form a hierarchy in terms of the types of redundancy they avoid
  - 1NF - attribute domains have to be atomic
  - 2NF - non-prime attributes do not depend on parts of a key
  - 3NF - no non-prime attribute depends transitively on a key
  - BCNF - every attribute only depends on a candidate key
  - 4NF and 5NF (we will only briefly discuss these)

Normalforms & Decomposition Algorithms

Atomic Domains

Atomic Domains

- An attribute domain is atomic if its values can be considered as indivisible
  - not atomic: set-valued attributes, composite attributes
  - atomic: numbers, strings (sometimes)

When Are Domains Atomic?

Remark

- Atomicity is not a precise formal concept
- **rule of thumb**: if we do not need to divide the value into smaller parts, then we can consider it to be atomic

- Consider student ids that consists of a two characters for the major followed by a number. Is this atomic?
  - If we extract student majors from these ids then we should not consider them atomic
  - If we only use the complete values then we can consider student ids to be atomic

First Normal Form (1NF)

Definition (First Normal Form (1NF))

A relation R is in 1NF if the domains of all attributes in **R** are atomic

Redundancy caused by non-atomic values

- Consider encoding Address information as a string in a set-valued attribute

Name	Address
Peter	{ "456 Tyler St, Chicago", "3400 Michigan Ave, Chicago" }
Alice	{ "456 Tyler St, Chicago" }
Bob	{ "3400 Michigan Ave, Chicago" }

Normalforms & Decomposition Algorithms

Non-prime Attributes

Definition (Non-prime Attributes)

- Let $\text{CandKeys}(\mathbf{R}, \Sigma)$ denote the set of all **candidate keys** for **R**
- An attribute A is non-prime if:

$$\nexists K \in \text{CandKeys}(\mathbf{R}, \Sigma) : A \in K$$

- Let $\text{NonPrime}(\mathbf{R}, \Sigma)$ denote the set of **non-prime** attributes of **R**

Non-prime Attributes Example

Example

- $\mathbf{R}(A, B, C)$ with $\Sigma = \{A \to B; B \to C\}$
- $\text{CandKeys}(\mathbf{R}, \Sigma) = \{\{A\}\}$, i.e., $\{A\}$ is the only candidate key
- $\Rightarrow B$ and C are non-prime

Second Normal Form (2NF)

Definition (Second Normal Form (2NF))

A relation is in second normal form (2NF) iff

- It is in **1NF**
- **and** no non-prime attribute depends on parts of a candidate key:

$$\forall A \in \text{NonPrime}(\mathbf{R}, \Sigma) : \nexists \alpha \subset K \in \text{CandKeys}(\mathbf{R}, \Sigma) : \alpha \to A \in \Sigma^+$$

2NF Example

- $\mathbf{R}(A, B, C, D)$
  - $A, B \to C, D$
  - $A \to C$
  - $B \to D$

- $K = \{A, B\}$ is the only candidate key
- **R** is **not in 2NF**
  - $A \to C$ where $A \subset K$ and $C \in \text{NonPrime}(\mathbf{R}, \Sigma)$
- For instance, a more concrete interpretation of **R** is **Advisor**(*InstrSSN*, *StudentUIN*, *InstrName*, *StudentName*)
- This is an indication that we are putting stuff together that does not belong together

Why Is Non-2NF Bad?

- **Why is a dependency on parts of a candidate key bad?**
  - That is: Why is not being in 2NF bad?
- **Redundancy**

- **Advisor** (InstrSSN, StudentCWID, InstrName, StudentName)
- *StudentCWID* $\to$ *StudentName*
- If a student has more than one adviser then the student's name will be repeated

- **Disconnect**
  - Some attributes are unrelated to parts of a candidate key
  - Indication that we have put an **N-M** relationship into a table including the attributes of the involved entities. We should decompose the relation.

Does 2NF Avoid All Types of Redundancy?

- **instructor** (name, salary, depname, depbudget) = $\mathbf{I}(A, B, C, D)$
- $\{Name\}$ is the only candidate key
- **I** is in **2NF**
- **Redundancy**
  - `depbudget` is repeated if there are more than one instructor in the same department

Normalforms & Decomposition Algorithms

Third Normal Form (3NF)

Definition (Third Normal Form (3NF))

A relation **R** with FDs Σ is in third normal form (3NF) if for **all** $\sigma : \alpha \rightarrow \beta \in \Sigma^+$ at **least one** of the following conditions holds:

1. $\alpha \rightarrow \beta$ is **trivial** ($\beta \subseteq \alpha$)
2. α is a **superkey**
3. each attribute $A \in (\beta - \alpha)$ is part of a some candidate key of **R**:

$$\forall A \in (\beta - \alpha) : \exists K \in \text{CandKeys}(\mathbf{R}, \Sigma) : A \in K$$

Remark

In the 3rd condition each attribute A may belong to a different candidate key!

Alternative Definition of 3NF

Alternative Interpretation

- Every **non-prime attribute** only depends **directly** on a **candidate key**

3NF Example

- **instructor** (name, salary, depname, depbudget) = $\mathbf{I}(A, B, C, D)$
- $\{Name\}$ is the only candidate key
- **I** is in **2NF**
- **I** is **not** in **3NF**

Testing for 3NF

Naive Algorithm

- Compute all candidate keys
- Compute Σ^+
- For each $\sigma \in \Sigma$ check whether one of the three conditions holds

Optimizations

- It is sufficient to check the conditions of 3NF on FDs in Σ instead of Σ^+
- Use attribute closure to determine whether α is a superkey for each FD $\alpha \rightarrow \beta \in \Sigma$
- If α is not a superkey then we need to check whether each attribute $\beta - \alpha$ is part of candidate key

Testing for 3NF - Computational Complexity

Computational Complexity

- Testing for 3NF is computationally hard (**NP-hard**)
- Why? Computing candidate keys is hard

Blind Decomposition

- Given the computational complexity, it is **not practical** to test whether relations with many attribute and / or many FDs are in **3NF**
- Should we just give up on 3NF?
- No! There exists a decomposition algorithm that takes a relation schema **R** and creates lossless join decomposition $\mathbf{R_1}, \ldots, \mathbf{R_n}$ of **R** such that every $\mathbf{R_i}$ is in 3NF

Decomposition Algorithm

Algorithm 7: 3NF

Input : Relation **R** with canonical cover Σ_C of FDs Σ
Output: Decomposition $\mathbf{R_1}, \ldots \mathbf{R_n}$

$result \leftarrow \emptyset, i \leftarrow 0$
for $\sigma : \alpha \rightarrow \beta \in \Sigma_C$ **do**
  if $\nexists j \in [1, i-1] : (\alpha \cup \beta) \subseteq \mathbf{R_j}$ **then** /* ensure one fragment contains FD's attributes */
    $i \leftarrow i + 1$
    $\mathbf{R_i} = \alpha \cup \beta$
    $result \leftarrow result \cup \{\mathbf{R_i}\}$
if $\nexists K \in \text{CandKeys}(\mathbf{R}, \Sigma) : \exists j \in [1, i] : K \subseteq \mathbf{R_j}$ **then** /* one fragment should have candidate key */
  $i \leftarrow i + 1$
  $\mathbf{R_i} = K$ for some $K \in \text{CandKeys}(\mathbf{R}, \Sigma)$
while $\exists \mathbf{R_j}, \mathbf{R_k} \in result : \mathbf{R_j} \subseteq \mathbf{R_k}$ **do** /* remove redundant relations */
  $result \leftarrow result - \{\mathbf{R_j}\}$
return $result$

Properties of the Decomposition Algorithm

- The algorithm is in **PTIME**
- The decomposition $\mathbf{R_1}, \ldots, \mathbf{R_n}$ computed by the algorithm has the following properties
  - each $\mathbf{R_i}$ is in **3NF**
  - the decomposition is **dependency preserving** and **lossless-join**

Paradox?

- Does the existence of a PTIME algorithm for decomposition contradict the hardness of the 3NF testing problem?
  - Why can't we apply the decomposition algorithm to **R** and if the algorithm does not decompose **R** then **R** was already in 3NF?
- We can reconcile these two results by observing that the algorithm may sometimes further decompose a relation that is already in 3NF
  - Thus, we cannot use it to test for 3NF

3NF Decomposition Example

- **cust_banker_branch** (customer_id, employee_id, banch_name, type)

$$\Sigma = \{$$
$$\sigma_1 : customer\_id, employer\_id \rightarrow branch\_name, type$$
$$\sigma_2 : employee\_id \rightarrow branch\_name$$
$$\sigma_3 : customer\_id, branch\_name \rightarrow employee\_id$$
$$\}$$

3NF Decomposition Example - Compute Canonical Cover

- **(1) Compute a canonical cover**
  - `branch_name` is extraneous in the RHS of σ_1
  - no other attribute is extraneous, so:

$$\Sigma_C = \{$$
$$\sigma_1' : customer\_id, employee\_id \rightarrow type$$
$$\sigma_2 : employee\_id \rightarrow branch\_name$$
$$\sigma_3 : customer\_id, branch\_name \rightarrow employee\_id$$
$$\}$$

3NF Decomposition Example - Decomposition

- **(2) Ensure that each FD's attributes appear together in one or more fragments**
  - fragments created in this step
    - $\mathbf{R_1}$(customer_id, employee_id, type)
    - $\mathbf{R_2}$(customer_id, branch_name)
    - $\mathbf{R_3}$(customer_id, branch_name, employee_id)
- **(3) Ensure that at least one fragment contains a candidate key**
  - $\mathbf{R_1}$ contains the candidate key $\{customer\_id, employee\_id\}$
  - no additional fragments have to be added in this step
- **(4) Remove contained fragments**
  - $\mathbf{R_2}$ is contained in $\mathbf{R_3}$, $\mathbf{R_2}$ will be removed
- **(5) Final result**
  - $\mathbf{R_1}$(customer_id, employee_id, type)
  - $\mathbf{R_3}$(customer_id, branch_name, employee_id)

Redundancy in 3NF

- **R** (S,I,D)
- $\Sigma = \{S, D \rightarrow I, I \rightarrow D\}$

S	I	D
s_1	i_1	d_1
s_2	i_1	d_1
s_3	i_1	d_1
s_3	i_2	d_2

- **dept_advisor** (studentid, instructorid, dept_name)
  - instructors work for one department only
  - a student has a unique advisor from each department
- Candidate keys are $\{S, D\}$ and $\{S, I\}$
- This relation is in **3NF**, but exhibits redundancy:
  - if an instructor appears in multiple tuples, then the department is repeated, e.g., (i_1, d_1)

Normalforms & Decomposition Algorithms

Boyce-Codd Normal Form (BCNF)

Definition (Boyce-Codd Normal Form (BCNF))

A relation schema **R** with FDs Σ is in Boyce-Codd Normal Form if for every functional dependency $\sigma \in \Sigma^+$ at least one of the following conditions holds:

- $\alpha \rightarrow \beta$ is trivial
- α is a superkey for **R**, i.e., $\alpha \rightarrow \mathbf{R} \in \Sigma^+$

- **inst_dept** (ID, name, salary, dept_name, building, budget)
  - with $\sigma : dept\_name \rightarrow building, budget$ in Σ
- This relation is not in **BCNF** as *dept_name* is not a superkey and the FD σ is not trivial

Testing for BCNF

Testing for BCNF

- For each FD $\sigma : \alpha \rightarrow \beta \in \Sigma^+$ check whether it fulfills one of the two conditions
  - $\beta \subseteq \alpha$
  - $\alpha^+ = \mathbf{R}$ (α is a superkey)

Optimizations

- It can be shown that it suffices to test only the FDs in Σ
- $\Rightarrow$ **testing for BCNF is in PTIME**

Testing for BCNF after Decomposition

Caveat

- The **optimization is only applicable on the original relation before decomposition!**
- **Testing whether the dependencies are preserved is computationally hard!**

- Consider **R** (A,B,C,D,E) with $\Sigma = \{A \rightarrow B, B, C \rightarrow D\}$
  - Decompose **R** into $\mathbf{R_1}$(A,B) and $\mathbf{R_2}$(A,C,D,E)
  - None of the original FDs contain only attributes from $\mathbf{R_2}$ so $\Sigma_2 = \emptyset$
    - Applying the optimized test to $\mathbf{R_2}$ would mislead us to think that this fragment is in BCNF
  - However, $A, C \rightarrow D \in \Sigma^+$ based on which $\mathbf{R_2}$ is not in BCNF

Decomposition Algorithm

Algorithm 8: BCNF Decomposition

Input : Relation **R** with FDs Σ
Output: Decomposition $\mathbf{R_1}, \ldots \mathbf{R_n}$

```
result ← R, i ← 0, done = false
while ¬ done do
    if ∃i : R_i not in BCNF then                    /* one fragment not in BCNF */
        Let σ = α → β such that α → R_i ∉ Σ⁺ ∧ α ∩ β = ∅
        result ← (result − R_i) ∪ {(R_i − β), (α ∪ β)}
    else
        done = true
return result
```


Properties of the Decomposition Algorithm

Runtime Complexity

- The algorithm is exponential time because of the potential need to compute Σ^+
- There are PTIME algorithms for BCNF decomposition, but ...
  - as for 3NF they may decompose more than necessary

Lossless Join Decomposition

- The algorithm guarantees that the decomposition is lossless
  - When we split a fragment we produce $\mathbf{R}_j = \mathbf{R}_i - \beta$ and $\mathbf{R}_k = \alpha \cup \beta$ based on an FD $\alpha \rightarrow \beta$.
  - As $\mathbf{R}_j \cap \mathbf{R}_k = \alpha$ the FD $\mathbf{R}_j \cap \mathbf{R}_k \rightarrow \mathbf{R}_k$ holds which means that the decomposition is lossless

BCNF and Dependency Preservation

Theorem (Impossibility of Dependency Preservation)

There exists a schema **R** *and set of FDs* Σ *such that there exists no BCNF decomposition of* **R** *that is dependency preserving*

$\mathbf{R} = (J, K, L)$

$$\Sigma = \{ \quad J, K \rightarrow L \quad L \rightarrow K \quad \}$$

- Two candidate keys $\{J, K\}$ and $\{J, L\}$
- **R** is not in BCNF
- Any decomposition of **R** that is in BCNF will fail to preserve: $J, K \rightarrow L$

Does BCNF Solve All of Our Problems with Redundancy?

- There are schemas in BCNF that still exhibit redundancy

- instructors can have multiple children and phone numbers
- id 1 has children (*Bob* and *Pete*) and phone numbers (*312-888-8888* and *312-777-5555*)

InstrID	child	phone
1	Pete	312-888-8888
1	Pete	312-777-5555
1	Bob	312-888-8888
1	Bob	312-777-5555

- Only trivial functional dependencies hold on this relation
- Redundancy stems from the independence of children and phone numbers
  — Adding another phone number we have to insert one tuple per child

Does BCNF Solve All of Our Problems with Redundancy?

The redundancy in this example can be solved using decomposition:

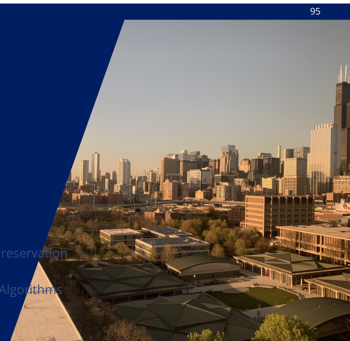
InstrID	child
1	Pete
1	Bob

InstrID	phone
1	312-888-8888
1	312-777-5555

Additional Normal Forms

- Removing further redundancies requires more powerful types of constraints and further normal forms
  — multivalued dependencies and join dependencies
  — 4NF
  — 5NF or Project-join Normal Form
  — Domain-key Normal Form (DKNF)

Recap



Recap

- Functional dependencies and other constraints
  — Armstrong's Axioms
  — Inference
  — Closure and attribute closure
  — Canonical Cover
- Redundancy & lossless join decomposition
- Normal Forms
  — 1NF, 2NF, 3NF, BCNF (and higher normal forms)
  — Testing for normal forms and decomposition algorithms

Bibliography

Bibliography

[1] C. Beeri and P.A. Bernstein.
Computational problems related to the design of normal form relational schemas.
ACM Transactions on Database Systems (TODS), 4(1):30–59, 1979.

Appendix

Appendix

Multivalued Dependencies

Definition (Multivalued Dependency)

The multivalued dependency (MVD) $\alpha \twoheadrightarrow \beta$ holds on R iff for any pair of tuples t_1 and t_2 with $t_1[\alpha] = t_2[\alpha]$ there exists two tuples t_3 and t_4 in R such that

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[\mathbf{R} - \beta] = t_2[\mathbf{R} - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_3[\mathbf{R} - \beta] = t_1[\mathbf{R} - \beta]$$

Remarks

FDs imply MVDs

Consider a schema **R** and $\alpha \subseteq$ **R** and $\beta \subseteq$ **R**, then

$$\alpha \rightarrow \beta \Rightarrow \alpha \twoheadrightarrow \beta$$

Trivial MVDs

- An MVD σ is trivial if $\emptyset \Rightarrow \sigma$.
- An MVD $\alpha \twoheadrightarrow \beta$ is trivial if either:
  - $\beta \subseteq \alpha$
  - $\mathbf{R} = \alpha \cup \beta$

MVD Example

- Let us revisit the the example in BCNF that still exhibited redundancy

- instructors can have multiple children and phone numbers
- id 1 has children (*Bob* and *Pete*) and phone numbers (*312-888-8888* and *312-777-5555*)

InstrID	child	phone
1	Pete	312-888-8888
1	Pete	312-777-5555
1	Bob	312-888-8888
1	Bob	312-777-5555

- MVDs:

$\Sigma = \{ID \twoheadrightarrow child; ID \twoheadrightarrow phone\}$

- For any two tuples $t_1 = (i, c_1, p_1)$ and $t_2 = (i, c_2, p_2)$ there also exists:
  - $t_3 = (i, c_1, p_2)$ and $t_4 = (i, c_2, p_1)$

Appendix

Fourth Normal Form (4NF)

Definition (4NF)

A relation **R** with functional and multivalued dependencies Σ is in 4NF if for every multivalued dependency is one of the two conditions hold:

1. $\alpha \twoheadrightarrow \beta$ is a trivial multivalued dependency
2. α is a superkey of **R**

Remark

- 4NF is stricter than BCNF
- Why? Because FDs imply MVDs but not necessarily vice versa

4NF and Redundancy

- A relation in 4NF may still exhibit redundancies that can be fixed through decomposition

agent	product	company
Bob	Laptop	ABM
Bob	Memory	ABM
John	Laptop	Pear
John	Memory	Pear
Pete	Disk	ABM
Pete	Disk	X
Pete	Laptop	ABM
Pete	Laptop	Pear

- No non-trivial FDs and MVDs hold on this relation
- It is in **4NF**
- Note that **R** can be decomposed into
  - $\mathbf{R}_1 = (agent, product)$
  - $\mathbf{R}_2 = (agent, company)$
  - $\mathbf{R}_3 = (product, company)$

Appendix

Definition (Join Dependency)

Consider a relation R with schema **R** and a decomposition $\mathbf{R}_1, \ldots, \mathbf{R}_n$. The relation fulfills the join dependency (JD) $\bowtie(\mathbf{R}_1, \ldots, \mathbf{R}_n)$ iff:

$$R = \pi_{\mathbf{R}_1}(R) \bowtie \ldots \bowtie \pi_{\mathbf{R}_n}(R)$$

Remark

- join dependencies are defined based on **lossless join decomposition**!
- join dependencies generalize MVDs as $\alpha \twoheadrightarrow \beta$ over $\mathbf{R} = \alpha \cup \beta \cup \gamma$ is equivalent to a binary join dependency $\bowtie(\alpha \cup \beta, \alpha \cup \gamma)$

Inference

- The inference problem for join dependencies is decidable
- However, there does not exist a **sound** and **complete** axiomatization for join dependencies

Appendix

Fifth Normal Form (5NF)

Definition (5NF)

Let Σ be a set of FDs, MVDs, and JDs for a relation **R** and let Δ denote all the key dependencies of **R**, i.e., FDs of the form $\alpha \rightarrow$ **R** where α is a candidate key. **R** is in project-join normal form also called fifth normal form if for every join dependency σ

$$(\Delta \Rightarrow \sigma) \Leftrightarrow \sigma \in \Sigma^{+}$$