CSE 544
Principles of Database Management Systems

Alvin Cheung
Fall 2015

Lecture 2 – SQL and Schema Normalization
Announcements

• Paper review
  – First paper review is due on Wednesday 10:30am
  – Details on website

• Find partners (0 or more) for the project
  – Project groups due on Friday (email)
  – You don’t need to choose a project yet; more suggestions will continue to be posted on website

• Homework 1 will be released by tomorrow!
  – Due in two weeks
Outline

Three topics today

• Wrap up relational algebra

• Crash course on SQL

• Brief overview of database design
Outline

Three topics today

• Wrap up relational algebra

• Crash course on SQL

• Brief overview of database design
Relational Operators

- **Selection**: $\sigma_{\text{condition}}(S)$
  - Condition is Boolean combination ($\land, \lor$) of terms
  - Term is: attr. op constant, attr. op attr.
  - Op is: $<$, $\leq$, $=$, $\neq$, $\geq$, or $>$

- **Projection**: $\pi_{\text{list-of-attributes}}(S)$

- **Union** ($\cup$), **Intersection** ($\cap$), **Set difference** ($-$),
- **Cross-product** or **cartesian product** ($\times$)
- **Join**: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
- **Division**: $R/S$
- **Rename**: $\rho(R(F), E)$
# Cross-Product Example

AnonPatient $P$

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

Voters $V$

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

$P \times V$

<table>
<thead>
<tr>
<th>P.age</th>
<th>P.zip</th>
<th>disease</th>
<th>name</th>
<th>V.age</th>
<th>V.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>
Join Galore

- **Theta-join**: \( R \bowtie_\theta S = \sigma_\theta(R \times S) \)
  - Join of \( R \) and \( S \) with a join condition \( \theta \)
  - Cross-product followed by selection \( \theta \)

- **Equijoin**: \( R \bowtie_\theta S = \pi_A (\sigma_\theta(R \times S)) \)
  - Join condition \( \theta \) consists only of equalities
  - Projection \( \pi_A \) drops all redundant attributes

- **Natural join**: \( R \bowtie S = \pi_A (\sigma_\theta(R \times S)) \)
  - aka Equijoin
  - Equality on all fields with same name in \( R \) and in \( S \)
### Theta-Join Example

#### AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>19</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

#### Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

\[ P \bowtie_{P.zip = V.zip \text{ and } P.age \leq V.age + 1 \text{ and } P.age \geq V.age - 1} V \]

\[
\begin{array}{ccc|ccc}
P.age & P.zip & disease & name & V.age & V.zip \\
19    & 98120 & flu      & p2   & 20    & 98120  \\
\end{array}
\]
Equijoin Example

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

\[ P \bowtie_{P.age = V.age} V \]

<table>
<thead>
<tr>
<th>age</th>
<th>P.zip</th>
<th>disease</th>
<th>name</th>
<th>V.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>98120</td>
</tr>
</tbody>
</table>
### Natural Join Example

<table>
<thead>
<tr>
<th></th>
<th>AnonPatient P</th>
<th>Voters V</th>
</tr>
</thead>
<tbody>
<tr>
<td>age</td>
<td>zip</td>
<td>name</td>
</tr>
<tr>
<td>54</td>
<td>98125</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td></td>
</tr>
</tbody>
</table>

\[ P \bowtie V \]

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
</tr>
</tbody>
</table>
Even More Joins

- **Outer join**
  - Include tuples with no matches in the output
  - Use NULL values for missing attributes

- **Variants**
  - Left outer join
  - Right outer join
  - Full outer join
### Outer Join Example

#### AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
</tr>
</tbody>
</table>

#### Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

#### $P \bowtie V$

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
<td>null</td>
</tr>
</tbody>
</table>
Example of Algebra Queries

Relations

Supplier(sno, sname, scity, sstate)
Part(pno, pname, psize, pcolor)
Supply(sno, pno, qty, price)

Q2: Name of supplier of parts with size greater than 10
\[ \pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{psize>10} (\text{Part}))) \]

Q3: Name of supplier of red parts or parts with size greater than 10
\[ \pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{psize>10} (\text{Part}) \cup \sigma_{pcolor='red'} (\text{Part}))) \]

(Many more examples in R&G)
Logical Query Plans

An RA expression but represented as a tree

Relations are sets of tuples
Each operator takes relations as input and outputs a relation
Can easily compose operators into expressions also called plans
Extended Operators of Relational Algebra

• **Duplicate elimination (δ)**
  - Since commercial DBMSs operate on **multisets/bags** not sets

• **Aggregate operators (γ)**
  - Useful in practice and requires bag semantics
  - Min, max, sum, average, count

• **Grouping operators (γ)**
  - Partitions tuples of a relation into “groups”
  - Aggregates can then be applied to groups

• **Sort operator (τ)**
Relational Calculus

- Alternative to relational algebra
  - Declarative query language
  - Describe what we want NOT how to get it
- Tuple relational calculus query
  - \{ T | p(T) \}
  - Where T is a tuple variable
  - p(T) denotes a formula that describes T
  - Result: set of all tuples for which p(T) is true
  - Language for p(T) is subset of first-order logic

Q1: Names of patients who have heart disease
\{ T | \exists P \in \text{AnonPatient} \exists V \in \text{Voter} \\
(P.zip = V.zip \land P.age = V.age \land P.disease = 'heart' \land T.name = V.name ) \}
Outline

Three topics today

• Wrap up relational algebra

• Crash course on SQL

• Brief overview of database design
Structured Query Language: SQL

• Influenced by relational calculus

• **Declarative query language**

• **Multiple aspects of the language**
  – Data definition language (DDL)
    • Statements to create, modify tables and views
  – Data manipulation language (DML)
    • Statements to issue queries, insert, delete data
  – More
Outline

• Today: crash course in SQL DML
  – Data Manipulation Language
  – SELECT-FROM-WHERE-GROUPBY
  – Study independently: INSERT/DELETE/MODIFY

• Study independently SQL DDL
  – Data Definition Language
  – CREATE TABLE, DROP TABLE, CREATE INDEX, CLUSTER, ALTER TABLE, …
  – E.g. google for the postgres manual, or type this in psql:
    \h create
    \h create table
    \h cluster
SQL Query

Basic form: (plus many many many more bells and whistles)

```
SELECT <attributes>
FROM <one or more relations>
WHERE <conditions>
```
Simple SQL Query

SELECT PName, Price, Manufacturer 
FROM Product 
WHERE Price > 100

“selection” and “projection”
Eliminating Duplicates

```
SELECT DISTINCT category
FROM Product
```

Compare to:

```
SELECT category
FROM Product
```

```
<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>

```

```
<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>

CSE 544 - Fall 2015
Ordering the Results

```
SELECT  pname, price, manufacturer
FROM    Product
WHERE   category='gizmo' AND price > 50
ORDER BY price, pname
```

Ties are broken by the 2\textsuperscript{nd} attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the \texttt{DESC} keyword.

Can also request only top-k with \texttt{LIMIT} clause
Joins

Product (pname, price, category, manufacturer)
Company (cname, stockPrice, country)

Find all products under $200 manufactured in Japan; return their names and prices.

```
SELECT P.pname, P.price
FROM Product P, Company C
WHERE P.manufacturer=C.cname AND C.country='Japan'
AND P.price <= 200
```

```
SELECT P.pname, P.price
FROM Product P JOIN Company C ON P.manufacturer=C.cname
WHERE C.country='Japan' AND P.price <= 200
```
Semantics of SQL Queries

```
SELECT a_1, a_2, ..., a_k
FROM   R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE  Conditions

Answer = {}
for x_1 in R_1 do
  for x_2 in R_2 do
    ....
    for x_n in R_n do
      if Conditions
      then Answer = Answer \cup \{(a_1,\ldots,a_k)\}
  return Answer
```
Aggregation

except count, all aggregations apply to a single attribute

SQL supports several aggregation operations:

sum, count, min, max, avg
Grouping and Aggregation

Purchase(product, price, quantity)

Find total quantities for all sales over $1, by product.

```
SELECT     product, Sum(quantity) AS TotalSales
FROM       Purchase
WHERE      price > 1
GROUP BY   product
```

Let’s see what this means…
Grouping and Aggregation

1. Compute the **FROM** and **WHERE** clauses.

2. Group by the attributes in the **GROUPBY**

3. Compute the **SELECT** clause: grouped attributes and aggregates.
1&2. FROM-WHERE-GROUPBY

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

WHERE price > 1
### 3. SELECT

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

```
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```
HAVING Clause

Same query as earlier, except that we consider only products that had at least 30 sales.

```sql
SELECT product, sum(price*quantity) FROM Purchase WHERE price > 1 GROUP BY product HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
WHERE vs HAVING

- **WHERE** condition is applied to individual rows
  - The rows may or may not contribute to the aggregate
  - No aggregates allowed here

- **HAVING** condition is applied to the entire group
  - Entire group is returned, or not at all
  - May use aggregate functions in the group
General form of Grouping and Aggregation

\[
\text{SELECT } S \\
\text{FROM } R_1, \ldots, R_n \\
\text{WHERE } C_1 \\
\text{GROUP BY } a_1, \ldots, a_k \\
\text{HAVING } C_2
\]

S = may contain attributes \( a_1, \ldots, a_k \) and/or any aggregates but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in \( R_1, \ldots, R_n \)

C2 = is any condition on aggregate expressions and on attributes \( a_1, \ldots, a_k \)
Semantics of SQL With Group-By

Evaluation steps:
1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes $a_1, \ldots, a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result
Subqueries

- A subquery is a SQL query nested inside a larger query
- Such inner-outer queries are called nested queries
- A subquery may occur in:
  - A SELECT clause
  - A FROM clause
  - A WHERE clause

- Rule of thumb: avoid writing nested queries when possible; keep in mind that sometimes it’s impossible
Subqueries in WHERE

Find all companies that make some products with price < 200

Using EXISTS:

```sql
SELECT DISTINCT C.cname
FROM Company C
WHERE EXISTS (SELECT *
  FROM Product P
  WHERE C.cid = P.cid AND P.price < 200)
```
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make \textit{some} products with price < 200

Using \textbf{IN}

\begin{verbatim}
SELECT DISTINCT C.cname
FROM Company C
WHERE C.cid IN (SELECT P.cid
FROM Product P
WHERE P.price < 200)
\end{verbatim}
Subqueries in WHERE

Find all companies that make some products with price < 200

Using **ANY**:  

```
SELECT DISTINCT C.cname  
FROM Company C  
WHERE 200 > ANY (SELECT price  
FROM Product P  
WHERE P.cid = C.cid)
```
Subqueries in WHERE

Find all companies that make some products with price < 200

Now let’s unnest it:

```
SELECT DISTINCT C.cname
FROM Company C, Product P
WHERE C.cid = P.cid and P.price < 200
```

Existential quantifiers are easy! 😊
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make only products with price < 200

same as:
Find all companies whose products all have price < 200

Universal quantifiers are hard! 😞
Subqueries in WHERE

1. Find the other companies: i.e. s.t. some product $\geq$ 200

```
SELECT DISTINCT C.cname
FROM Company C
WHERE C.cid IN (SELECT P.cid
    FROM Product P
    WHERE P.price $\geq$ 200)
```

2. Find all companies s.t. all their products have price < 200

```
SELECT DISTINCT C.cname
FROM Company C
WHERE C.cid NOT IN (SELECT P.cid
    FROM Product P
    WHERE P.price $\geq$ 200)
```
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make only products with price < 200

Using **EXISTS**:

```
SELECT DISTINCT C.cname
FROM Company C
WHERE NOT EXISTS (SELECT *
    FROM Product P
    WHERE P.cid = C.cid and P.price >= 200)
```
Subqueries in WHERE

Product (pname, price, cid)
Company(cid, cname, city)

Find all companies that make only products with price < 200

Using ALL:

```
SELECT DISTINCT C.cname
FROM Company C
WHERE 200 > ALL (SELECT price
FROM Product P
WHERE P.cid = C.cid)
```
Can we unnest the *universal quantifier* query?

- A query Q is **monotone** if:
  - Whenever we add tuples to one or more of the tables...
  - ... the answer to the query cannot contain fewer tuples

- **Fact**: all unnested queries are monotone
  - Proof: using the “nested for loops” semantics

- **Fact**: Query with universal quantifier is not monotone

- **Consequence**: we cannot unnest a query with a universal quantifier
Outline

Three topics today

• Wrap up relational algebra
• Crash course on SQL
• Brief overview of database design
Database Design

• The relational model is great, but how do I design my database schema?
Outline

• Conceptual db design: entity-relationship model

• Problematic database designs

• Functional dependencies

• Normal forms and schema normalization
Database Design Process

Data Modeling → Refinement → SQL Tables → Files

ER diagrams → Relations

Conceptual Schema

Physical Schema
Conceptual Schema Design

Conceptual Model:

Relational Model: plus FD’s
(FD = functional dependency)

Normalization: Eliminates anomalies
Entity-Relationship Diagram

Patient

- name
- zip

Doctor

- dno
- specialty
- name

Relationship sets
- patient_of

Attributes
- name

Entity sets
- Patient
Entity-Relationship Model

- Typically, each entity has a key

- ER relationships can include multiplicity
  - One-to-one, one-to-many, etc.
  - Indicated with arrows

- Can model multi-way relationships

- Can model subclasses

- And more...
Subclasses to Relations

Product

name

category

price

Sw.Product

isa

Ed.Product

isa

platforms

Software Product

Educational Product

platforms

Age Group

Other ways to convert are possible

CSE 544 - Fall 2015

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>99</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>49</td>
<td>photo</td>
</tr>
<tr>
<td>Toy</td>
<td>39</td>
<td>gadget</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>unix</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Age Group</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>toddler</td>
</tr>
<tr>
<td>Toy</td>
<td>retired</td>
</tr>
</tbody>
</table>
General approach to Translating Diagram into Relations

Normally translate as follows:
• Each entity set becomes a relation
• Each relationship set becomes a relation
  – Except many-one relationships. Can combine them with entity set.

One **bad way** to translate our diagram into relations
• **PatientOf** *(pno, name, zip, dno, since)*
• **Doctor** *(dno, dname, specialty)*
Outline

• Conceptual db design: entity-relationship model

• Problematic database designs

• Functional dependencies

• Normal forms and schema normalization
Problematic Designs

• Some db designs lead to **redundancy**
  – Same information stored multiple times

• Problems
  – Redundant storage
  – Update anomalies
  – Insertion anomalies
  – Deletion anomalies
Problem Examples

PatientOf

<table>
<thead>
<tr>
<th>pno</th>
<th>name</th>
<th>zip</th>
<th>dno</th>
<th>since</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1</td>
<td>98125</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>98125</td>
<td>3</td>
<td>2003</td>
</tr>
<tr>
<td>2</td>
<td>p2</td>
<td>98112</td>
<td>1</td>
<td>2002</td>
</tr>
<tr>
<td>3</td>
<td>p1</td>
<td>98143</td>
<td>1</td>
<td>1985</td>
</tr>
</tbody>
</table>

What if we want to insert a patient without a doctor? What if we want to delete the last doctor for a patient? Illegal as (pno,dno) is the primary key, cannot have nulls.

Redundant If we update to 98119, we get inconsistency
## Solution: Decomposition

<table>
<thead>
<tr>
<th>pno</th>
<th>name</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1</td>
<td>98125</td>
</tr>
<tr>
<td>2</td>
<td>p2</td>
<td>98112</td>
</tr>
<tr>
<td>3</td>
<td>p1</td>
<td>98143</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
<th>dno</th>
<th>since</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>2003</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2002</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>1985</td>
</tr>
</tbody>
</table>

Decomposition solves the problem, but need to be careful…
Lossy Decomposition

<table>
<thead>
<tr>
<th>Patient</th>
<th>PatientOf</th>
</tr>
</thead>
<tbody>
<tr>
<td>pno</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
</tr>
<tr>
<td>2</td>
<td>p2</td>
</tr>
<tr>
<td>3</td>
<td>p1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Decomposition can cause us to lose information!
Schema Refinement Challenges

• How do we know that we should decompose a relation?
  – Functional dependencies
  – Normal forms

• How do we make sure decomposition does not lose info?
  – Lossless-join decompositions
  – Dependency-preserving decompositions
Outline

• Conceptual db design: entity-relationship model

• Problematic database designs

• Functional dependencies

• Normal forms and schema normalization
Functional Dependency

• A functional dependency (FD) is an integrity constraint that generalizes the concept of a key

• An instance of relation R satisfies the FD: \( X \rightarrow Y \)
  – if for every pair of tuples \( t_1 \) and \( t_2 \)
  – if \( t_1.X = t_2.X \) then \( t_1.Y = t_2.Y \)
  – where \( X, Y \) are two nonempty sets of attributes in \( R \)

• We say that \( X \) determines \( Y \)

• FDs come from domain knowledge
FD Example

An FD holds, or does not hold on an instance:

<table>
<thead>
<tr>
<th>EmpID</th>
<th>Name</th>
<th>Phone</th>
<th>Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>E0045</td>
<td>Smith</td>
<td>1234</td>
<td>Clerk</td>
</tr>
<tr>
<td>E3542</td>
<td>Mike</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E1111</td>
<td>Smith</td>
<td>9876</td>
<td>Salesrep</td>
</tr>
<tr>
<td>E9999</td>
<td>Mary</td>
<td>1234</td>
<td>Lawyer</td>
</tr>
</tbody>
</table>

EmpID → Name, Phone, Position
Position → Phone
but not Phone → Position
FD Terminology

• FD’s are constraints
  – On some instances they hold
  – On others they do not

• If every instance of R will be one in which a given FD will hold, then we say that R satisfies the FD
  – If we say that R satisfies an FD F, we are stating a constraint on R

• FDs come from domain knowledge
Decomposition Problems

• FDs will help us identify possible redundancy
  – Identify redundancy and split relations to avoid it.

• Can we get the data back correctly?
  – **Lossless-join decomposition**

• Can we recover the FD’s on the ‘big’ table from the FD’s on the small tables?
  – **Dependency-preserving decomposition**
  – So that we can enforce all FDs without performing joins
Outline

• Conceptual db design: entity-relationship model

• Problematic database designs

• Functional dependencies

• Normal forms and schema normalization
Normal Forms

• Based on Functional Dependencies
  – 2nd Normal Form (obsolete)
  – 3rd Normal Form
  – Boyce Codd Normal Form (BCNF)

• Based on Multivalued Dependencies
  – 4th Normal Form

• Based on Join Dependencies
  – 5th Normal Form

We only discuss these two
A simple condition for removing anomalies from relations:

A relation R is in BCNF if:

If $A_1, ..., A_n \rightarrow B$ is a non-trivial dependency in $R$, then \{A$_1$, ..., A$_n$\} is a superkey for R

BCNF ensures that no redundancy can be detected using FD information alone
Our Example

PatientOf

<table>
<thead>
<tr>
<th>pno</th>
<th>name</th>
<th>zip</th>
<th>dno</th>
<th>since</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1</td>
<td>98125</td>
<td>2</td>
<td>2000</td>
</tr>
<tr>
<td>1</td>
<td>p1</td>
<td>98125</td>
<td>3</td>
<td>2003</td>
</tr>
<tr>
<td>2</td>
<td>p2</td>
<td>98112</td>
<td>1</td>
<td>2002</td>
</tr>
<tr>
<td>3</td>
<td>p1</td>
<td>98143</td>
<td>1</td>
<td>1985</td>
</tr>
</tbody>
</table>

pno,dno is a key, but pno → name, zip
BCNF violation so we decompose
Decomposition in General

\[ R(A_1, \ldots, A_n, B_1, \ldots, B_m, C_1, \ldots, C_p) \]

\[ R_1(A_1, \ldots, A_n, B_1, \ldots, B_m) \]

\[ R_2(A_1, \ldots, A_n, C_1, \ldots, C_p) \]

\[ R_1 = \text{projection of } R \text{ on } A_1, \ldots, A_n, B_1, \ldots, B_m \]

\[ R_2 = \text{projection of } R \text{ on } A_1, \ldots, A_n, C_1, \ldots, C_p \]

**Theorem** If \( A_1, \ldots, A_n \rightarrow B_1, \ldots, B_m \)

Then the decomposition is lossless

Note: don’t necessarily need \( A_1, \ldots, A_n \rightarrow C_1, \ldots, C_p \)
**BCNF Decomposition Algorithm**

**Repeat**
- choose $A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n$ that violates BCNF condition
- split $R$ into $R_1(A_1, \ldots, A_m, B_1, \ldots, B_n)$ and $R_2(A_1, \ldots, A_m, \text{[rest]})$
- continue with both $R_1$ and $R_2$

**Until** no more violations

Lossless-join decomposition: Attributes common to $R_1$ and $R_2$ must contain a key for either $R_1$ or $R_2$
BCNF and Dependencies

<table>
<thead>
<tr>
<th>Unit</th>
<th>Company</th>
<th>Product</th>
</tr>
</thead>
</table>

FD’s:  \( \text{Unit} \rightarrow \text{Company}; \quad \text{Company, Product} \rightarrow \text{Unit} \)

So, there is a BCNF violation, and we decompose.
BCNF and Dependencies

FD’s:  Unit → Company;  Company, Product → Unit
So, there is a BCNF violation, and we decompose.

In BCNF we lose the FD:  Company, Product → Unit
A simple condition for removing anomalies from relations:

A relation $R$ is in 3rd normal form if:

Whenever there is a nontrivial dep. $A_1, A_2, \ldots, A_n \rightarrow B$ for $R$, then $\{A_1, A_2, \ldots, A_n\}$ is a super-key for $R$, or $B$ is part of a key.
3NF Discussion

• 3NF decomposition v.s. BCNF decomposition:
  – Use same decomposition steps, for a while
  – 3NF may stop decomposing, while BCNF continues

• Tradeoffs
  – BCNF = no anomalies, but may lose some FDs
  – 3NF = keeps all FDs, but may have some anomalies
Summary

• **Database design is not trivial**
  – Use ER models
  – Translate ER models into relations
  – Normalize to eliminate anomalies

• **Normalization tradeoffs**
  – BCNF: no anomalies, but may lose some FDs
  – 3NF: keeps all FDs, but may have anomalies
  – Too many small tables affect performance