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: $\sigma_{condition}(S)$
  - Condition is Boolean combination ($\wedge$, $\vee$) of terms
  - Term is: attr. op constant, attr. op attr.
  - Op is: <, <=, =, $\neq$, >=, or >
- Projection: $\pi_{list\text{-}of\text{-}attributes}(S)$
- 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

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

P x V

P.age	P.zip	disease	name	V.age	V.zip
54	98125	heart	p1	54	98125
54	98125	heart	p2	20	98120
20	98120	flu	p1	54	98125
20	98120	flu	p2	20	98120

Join Galore

- **Theta-join**: $R \bowtie_\theta S = \sigma_\theta(R \times S)$
  – Join of R and S with a join condition θ
  – Cross-product followed by selection θ
- **Equijoin**: $R \bowtie_\theta S = \pi_A(\sigma_\theta(R \times S))$
  – Join condition θ consists only of equalities
  – Projection π_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

age	zip	disease
50	98125	heart
19	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie_{P.zip = V.zip \text{ and } P.age <= V.age + 1 \text{ and } P.age >= V.age - 1} V$

P.age	P.zip	disease	name	V.age	V.zip
19	98120	flu	p2	20	98120

Equijoin Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie_{P.age=V.age} V$

age	P.zip	disease	name	V.zip
54	98125	heart	p1	98125
20	98120	flu	p2	98120

Natural Join Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie V$

age	zip	disease	name
54	98125	heart	p1
20	98120	flu	p2

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

age	zip	disease
54	98125	heart
20	98120	flu
33	98120	lung

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \overset{o}{\bowtie} V$

age	zip	disease	name
54	98125	heart	p1
20	98120	flu	p2
33	98120	lung	null

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

$\pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{psize>10} (\text{Part}))$

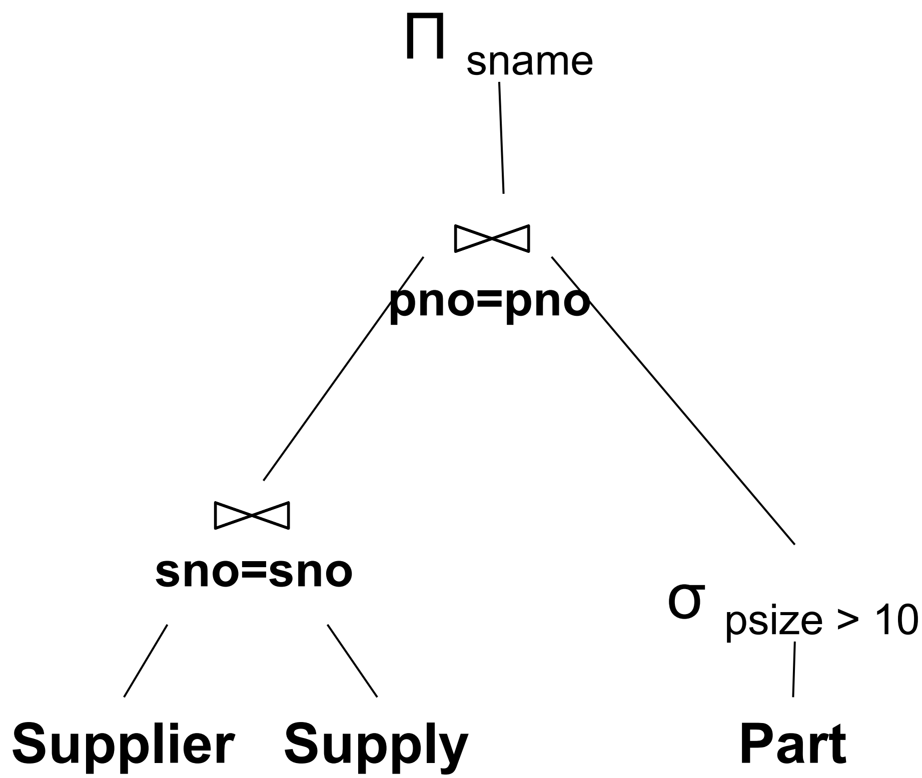
Q3: Name of supplier of red parts or parts with size greater than 10

$\pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{psize>10} (\text{Part}) \cup \sigma_{pcolor='red'} (\text{Part})))$

(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 | ∃ P ∈ AnonPatient ∃ V ∈ Voter

(P.zip = V.zip ∧ P.age = V.age ∧ P.disease = 'heart' ∧ 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

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

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

PName	Price	Manufacturer
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

"selection" and "projection"

Eliminating Duplicates

```
SELECT  DISTINCT category
FROM    Product
```

Category
Gadgets
Photography
Household

Compare to:

```
SELECT  category
FROM    Product
```

Category
Gadgets
Gadgets
Photography
Household

Ordering the Results

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

Ties are broken by the 2nd attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

Can also request only top-k with LIMIT clause

Joins

Product (<u>pname</u>, price, category, manufacturer)
Company (<u>cname</u>, 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 $a_1, a_2, \ldots, a_k$
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

```
SELECT  avg(price)
FROM    Product
WHERE   maker="Toyota"
```

```
SELECT  count(*)
FROM    Product
WHERE   year > 1995
```

SQL supports several aggregation operations:

sum, count, min, max, avg

Except count, all aggregations apply to a single attribute

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

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	~~0.5~~	~~50~~
Banana	2	10
Banana	4	10

WHERE price > 1

3. SELECT

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	~~0.5~~	~~50~~
Banana	2	10
Banana	4	10

Product	TotalSales
Bagel	40
Banana	20

What can go in SELECT clause?
Will return ONE TUPLE per group

```
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.

```
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 al all
  - May use aggregate functions in the group

General form of Grouping and Aggregation

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

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

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

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

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

Existential quantifiers

Find all companies that make <u>some</u> products with price < 200

Using EXISTS:

```
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

Existential quantifiers

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

Find all companies that make some products with price < 200

Using IN

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

Subqueries in WHERE

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

Existential quantifiers

Find all companies that make <u>some</u> 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

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

Existential quantifiers

Find all companies that make <u>some</u> 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

Universal quantifiers

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

Find all companies that make <u>only</u> products with price < 200

same as:

Find all companies whose products <u>all</u> have price < 200

Universal quantifiers are hard ! ☹

Subqueries in WHERE

1. Find *the other* companies: i.e. s.t. some product ≥ 200

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

Subqueries in WHERE

Universal quantifiers

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

Find all companies that make <u>only</u> 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)

Universal quantifiers

Find all companies that make <u>only</u> 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

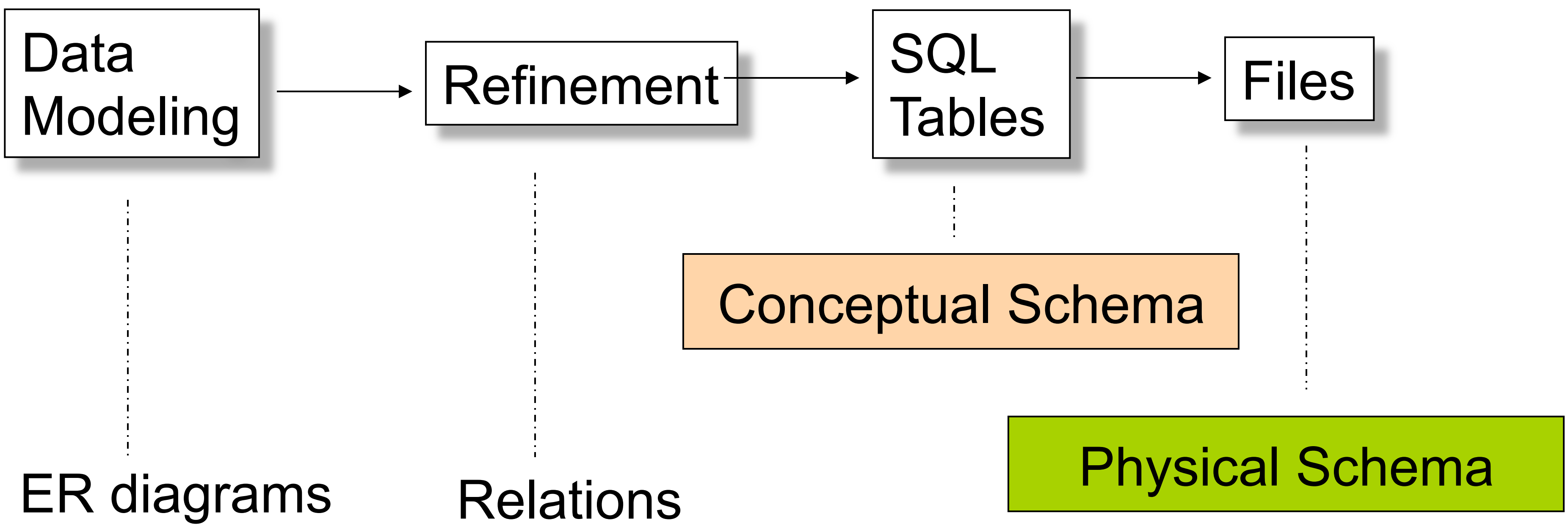
Database Design

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

Outline

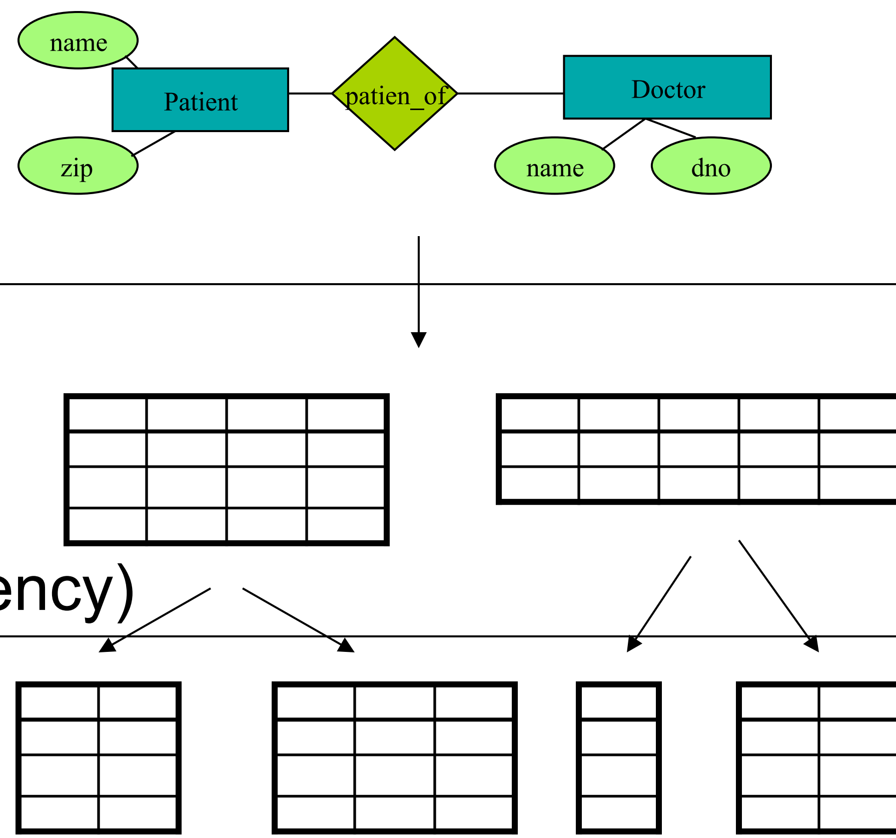
- Conceptual db design: entity-relationship model
- Problematic database designs
- Functional dependencies
- Normal forms and schema normalization

Database Design Process



Conceptual Schema Design

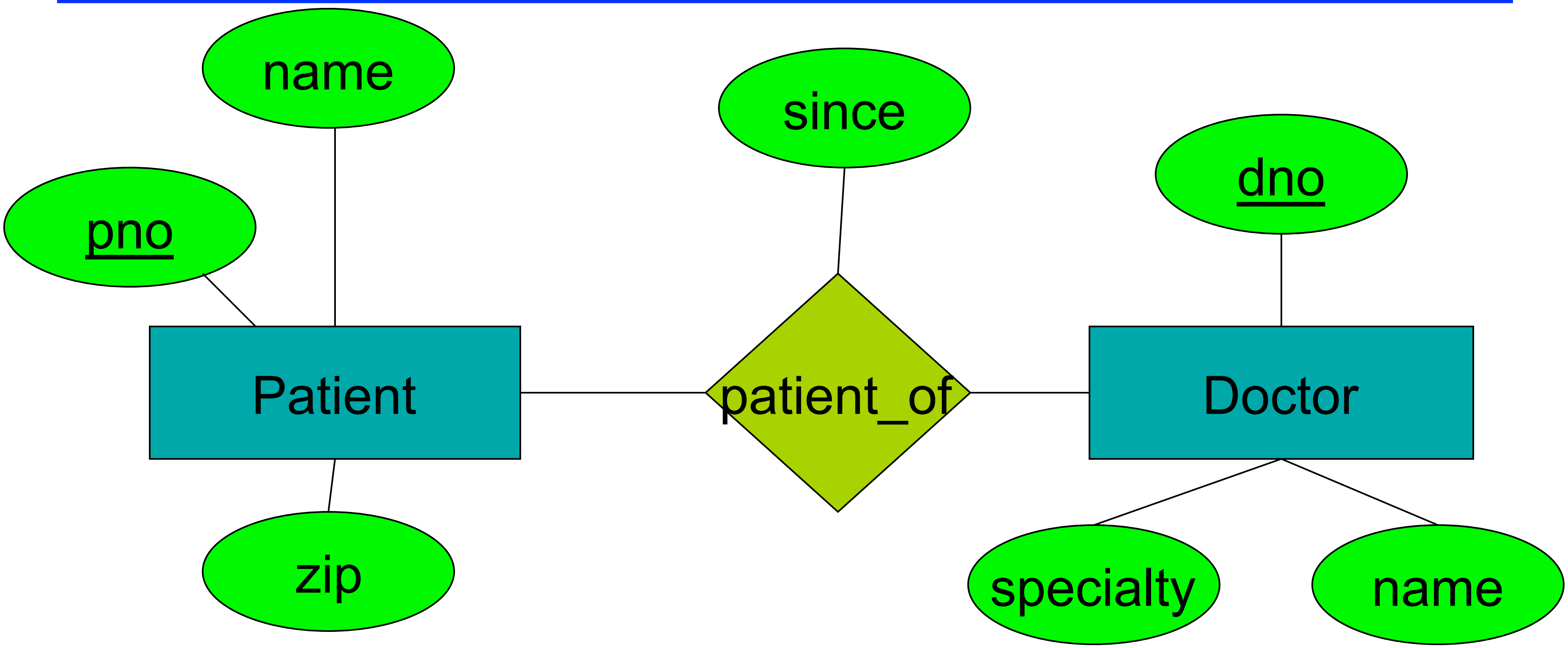
Conceptual Model:



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

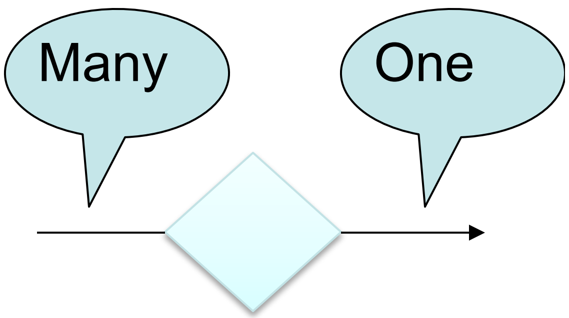
Normalization:
Eliminates anomalies

Entity-Relationship Diagram



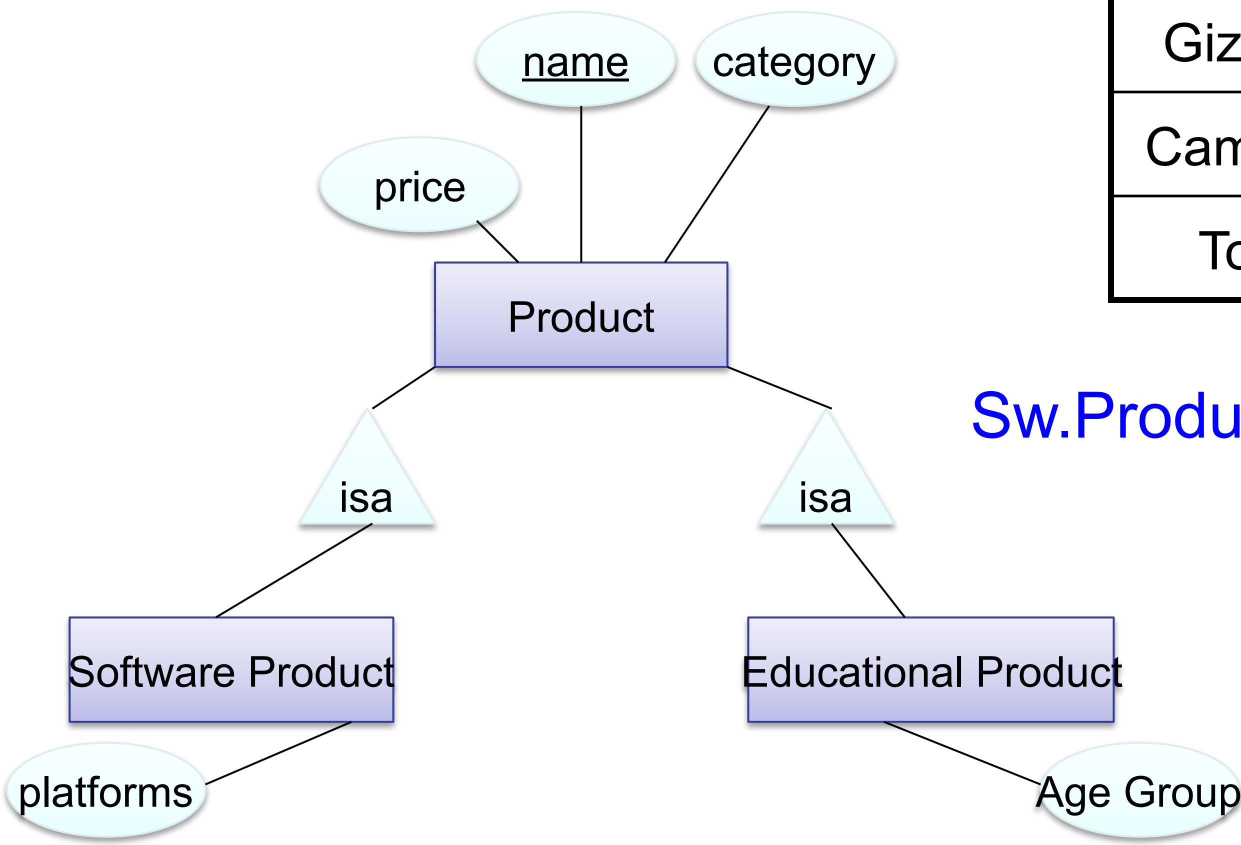
Entity-Relationship Model

- 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	Price	Category
Gizmo	99	gadget
Camera	49	photo
Toy	39	gadget

Sw.Product

Name	platforms
Gizmo	unix

Ed.Product

Name	Age Group
Gizmo	toddler
Toy	retired

Other ways to convert are possible

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 (<u>pno</u>, name, zip, <u>dno</u>, since)**
- **Doctor (<u>dno</u>, 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

pno	name	zip	dno	since
1	p1	98125	2	2000
1	p1	98125	3	2003
2	p2	98112	1	2002
3	p1	98143	1	1985

Redundant

If we update to 98119, we get inconsistency

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

Solution: Decomposition

Patient

pno	name	zip
1	p1	98125
2	p2	98112
3	p1	98143

PatientOf

pno	dno	since
1	2	2000
1	3	2003
2	1	2002
3	1	1985

Decomposition solves the problem,
but need to be careful…

Lossy Decomposition

Patient

pno	name	zip
1	p1	98125
2	p2	98112
3	p1	98143

PatientOf

name	dno	since
p1	2	2000
p1	3	2003
p2	1	2002
p1	1	1985

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 → Y**
  - if for every pair of tuples t1 and t2
  - if t1.X = t2.X then t1.Y = t2.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:

EmpID	**Name**	**Phone**	**Position**
E0045	Smith	1234	Clerk
E3542	Mike	9876	Salesrep
E1111	Smith	9876	Salesrep
E9999	Mary	1234	Lawyer

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)**

  } We only discuss these two

- Based on Multivalued Dependencies
  - 4th Normal Form
- Based on Join Dependencies
  - 5th Normal Form

BCNF

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

pno	name	zip	dno	since
1	p1	98125	2	2000
1	p1	98125	3	2003
2	p2	98112	1	2002
3	p1	98143	1	1985

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

Decomposition in General

$R(A_1, ..., A_n, B_1, ..., B_m, C_1, ..., C_p)$

↙ ↘

$R_1(A_1, ..., A_n, B_1, ..., B_m)$ $R_2(A_1, ..., A_n, C_1, ..., C_p)$

R_1 = projection of R on $A_1, ..., A_n, B_1, ..., B_m$
R_2 = projection of R on $A_1, ..., A_n, C_1, ..., C_p$

Theorem If $A_1, ..., A_n \rightarrow B_1, ..., B_m$
Then the decomposition is lossless

Note: don't necessarily need $A_1, ..., A_n \rightarrow C_1, ..., C_p$

BCNF Decomposition Algorithm

Repeat
choose $A_1, \ldots, A_m \rightarrow B_1, \ldots, B_n$ that violates BCNF condition
split R into

$$R_1(A_1, \ldots, A_m, B_1, \ldots, B_n) \text{ and } R_2(A_1, \ldots, A_m, [rest])$$

continue with both R1 and R2
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

Unit	Company	Product

FD's: Unit → Company; Company, Product → Unit

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

BCNF and Dependencies

Unit	Company	Product

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

Unit	Company

Unit → Company

Unit	Product

No FDs

In BCNF we lose the FD: Company, Product → Unit

3NF

A simple condition for removing anomalies from relations:

A relation R is in 3rd normal form if :

Whenever there is a nontrivial dep. $A_1, A_2, ..., A_n \rightarrow B$ for R, then $\{A_1, A_2, ..., A_n\}$ 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