What is a DBMS

- Database management systems:
  - Provide efficient and secure access to large amounts of data.
- Address problems such as:
  - How to store the data
  - How to query data efficiently
  - How to update the data securely (by multiple users)
- Contrast with using file systems for the same task

Relational Databases

- Based on the **relational model**

Student	Course	Quarter
Charles	CS 444	Fall, 1997
Dan	CS 142	Winter, 1998
…	…	…

- **Separates the logical view from the physical view of the data.**

Querying a Database

- Find all the students who have taken 94.301 in Fall 2001.
- S(tructured) Q(uery) L(anguage)
  - **select** E.name
  - **from** Enroll E
  - **where** E.course=94.301 and
  - E.quarter="Fall 2001"
- Query processor figures out how to answer the query efficiently.

Database Industry

- Relational databases are a great success of theoretical ideas.
- "Big 3" DBMS companies are among the largest software companies in the world.
- IBM (with DB2) and Microsoft (SQL Server, Microsoft Access) are also important players.
- $20B industry
- Challenged by object oriented DBMS.

Functionality of a DBMS

- Storage management
- Abstract data model
- High level query and data manipulation language
- Efficient query processing
- Transaction processing
- Resiliency: recovery from crashes
- Interface with programming languages

Why Use a DBMS?

- Data independence and efficient access.
- Reduced application development time.
- Data integrity and security.
- Uniform data administration
- Concurrent access and recovery from crashes.

The Study of DBMS

- Several aspects:
  - Modeling and design of databases
  - Database programming: querying and update operations
  - Database implementation
- DBMS study cuts across many fields of Computer Science: OS, languages, AI, Logic, multimedia, theory...

Database Design

- Why do we need it?
  - Agree on structure of the database before deciding on a particular implementation.
- Consider issues such as:
  - What entities to model
  - How entities are related
  - What constraints exist in the domain
  - How to achieve *good* designs

Database Design Formalisms

- Object Definition Language (ODL):
  - Closer in spirit to object-oriented models
- Entity/Relationship model (E/R):
  - More relational in nature.
- Both can be translated (semi-automatically) to relational schemas (with varying amount of pain).
- ODL to OO-schema: direct transformation (C++ or Smalltalk based system).

Object Definition Language

- Is part of ODMG, which also gave us OQL.
- Resembles C++ (and Smalltalk).
- Basic design paradigm in ODL:
  - Model ***objects*** and their ***properties.***
- For abstraction purposes:
  - Group objects into ***classes.***
- What qualifies as a ***good*** class?
  - Objects should have common properties.

ODL Class Declarations

Interface <name> {

attributes: <type> <name>;

relationships <range type> <name>;

methods

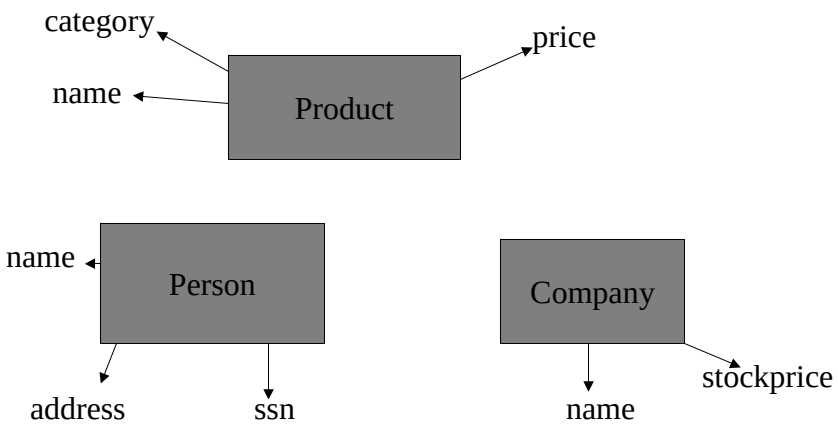
}

Method example:

float gpa(in: Student) **raises** (noGrades)

Arbitrary function can compute the value of gpa, based on a student object given as input.

ODL Example



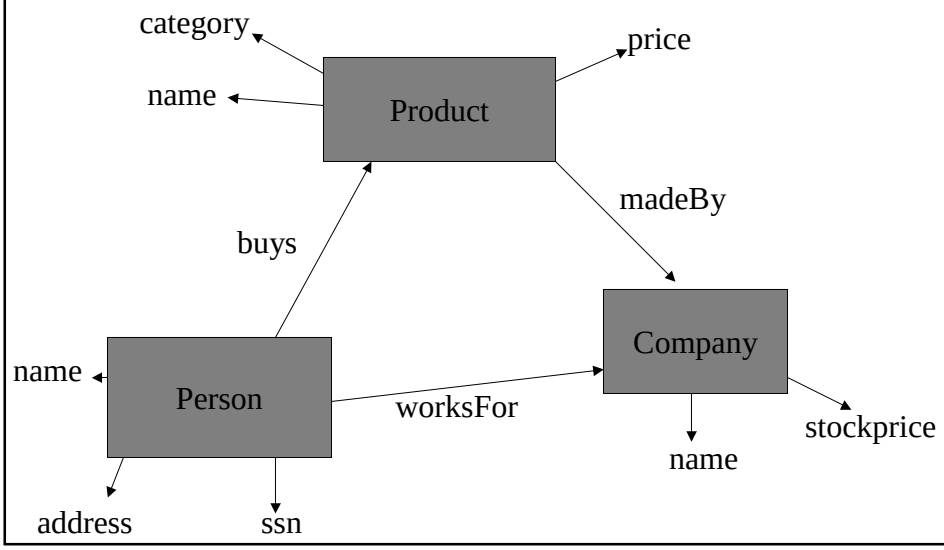
ODL Declarations

```
Interface Product {
        attribute  string name;
        attribute  float   price;
        attribute  enum Categories
                        {electronics, communications, sports …} category
        }

Interface Company {
        attribute   string  name;
        attribute   float    stockprice;
        }

Interface Person {
        attribute integer ssn;
        attribute string   name;
        attribute Struct Address {string street, string city} address; }
```

ODL Example

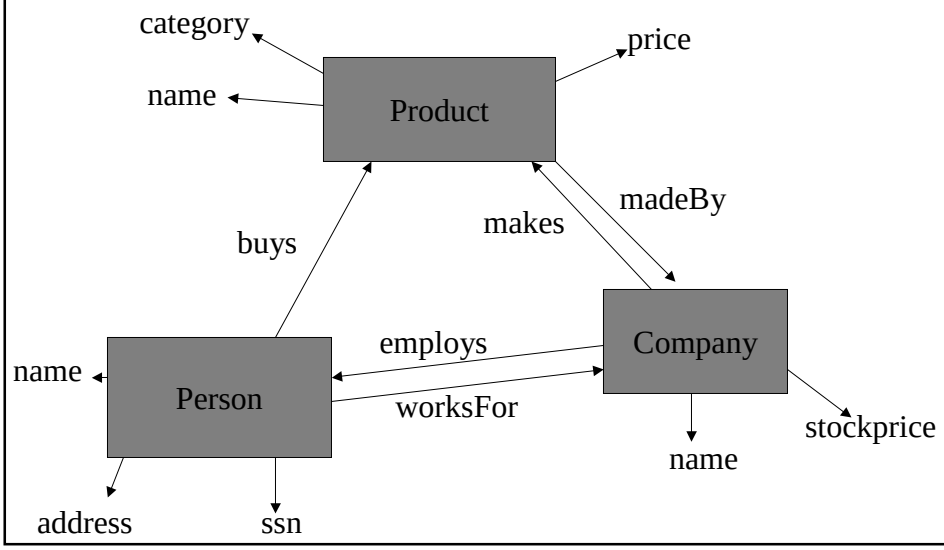


ODL Declarations

```
Interface Product {
        attribute  string name;
        attribute  float   price;
        attribute  enum Categories
                        {electronics, communications, sports …} category;
        relationship  <Company>  madeBy;
        }

Interface Person {
        attribute integer ssn;
        attribute string   name;
        attribute Struct Address {string street, string city} address;
        relationship  set <Product>  buys;
        relationship  set <Company>  worksFor;}
```

ODL Example



ODL Declarations

```
Interface Company {
        attribute   string  name;
        attribute   float    stockprice;

         relationship  set <Product> makes
                       inverse  Product::madeBy;

         relationship   set <Person>  employs
                       inverse  Person::worksFor;
        }
```

Types in ODL

Basic types:

Atomic types (e.g., string, integer, …)
Interface types (e.g., Person, Product, Company)

Constructors:

Set: (1, 5, 6)
Bag: (1, 1, 5, 6, 6)
List: (1, 5, 6, 1, 6)
Array: Integer[17]

Struct: {string street, string city, integer zipcode}

Allowable Types in ODL

For attributes: start with atomic or struct, and apply a collection type.

OK: string, set of integer, bag of Address.
Not OK: Product, set of set of integer.

For relationships:
start with interface type and apply a collection type.

OK: Product, set of Product, list of Person.
Not OK: struct {pname Product, cname Company}
set of bag of Product
integer

Entity / Relationship Diagrams

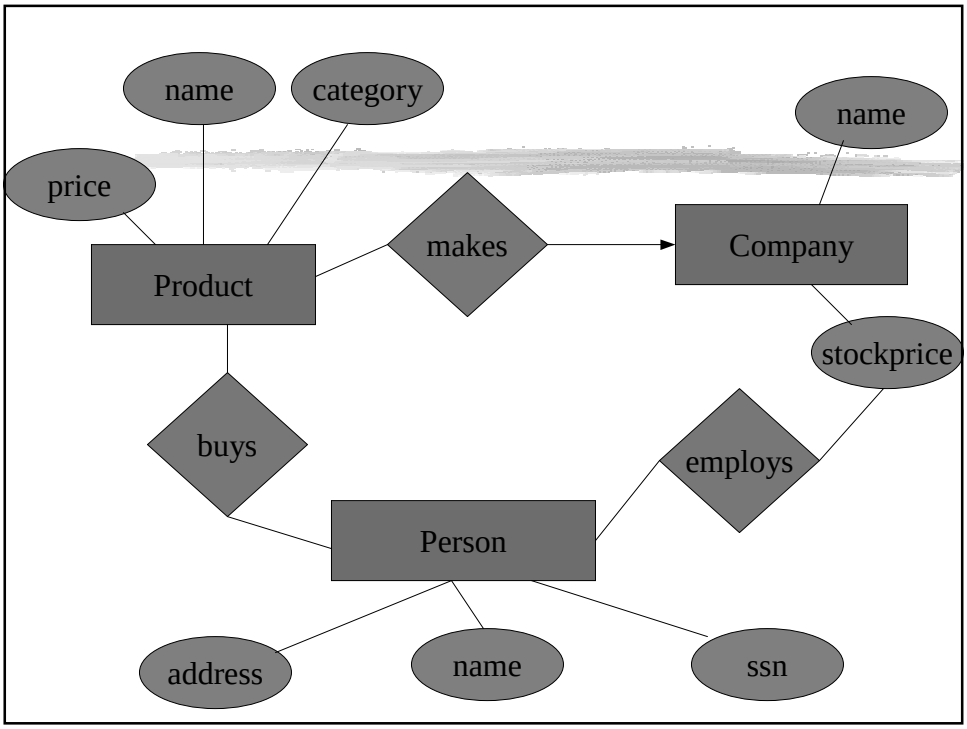
Objects → entities
Classes → entity sets

Attributes are like in ODL.

Relationships: like in ODL except

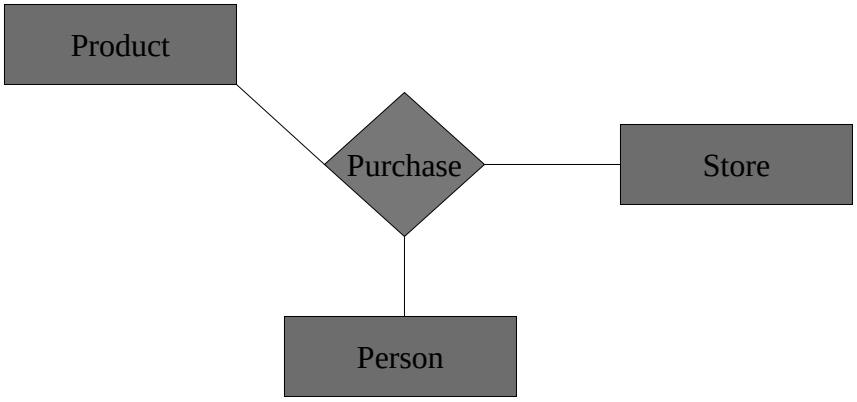
- not associated with classes (I.e., first class citizens)

- not necessarily binary



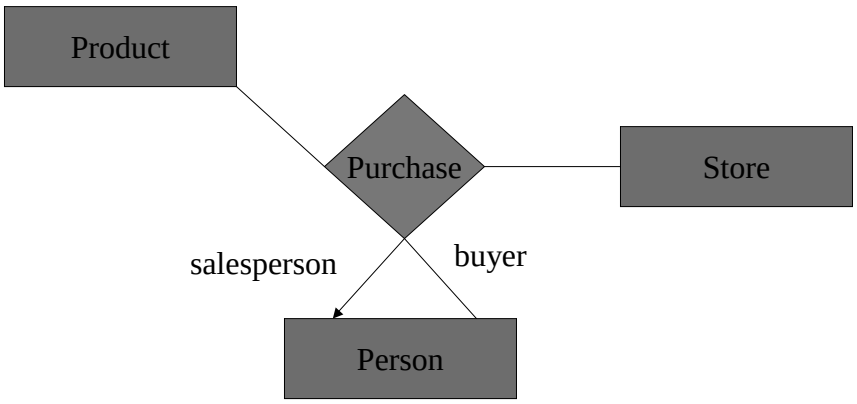
Multi-way Relationships

How do we model a purchase relationship between buyers, products and stores?

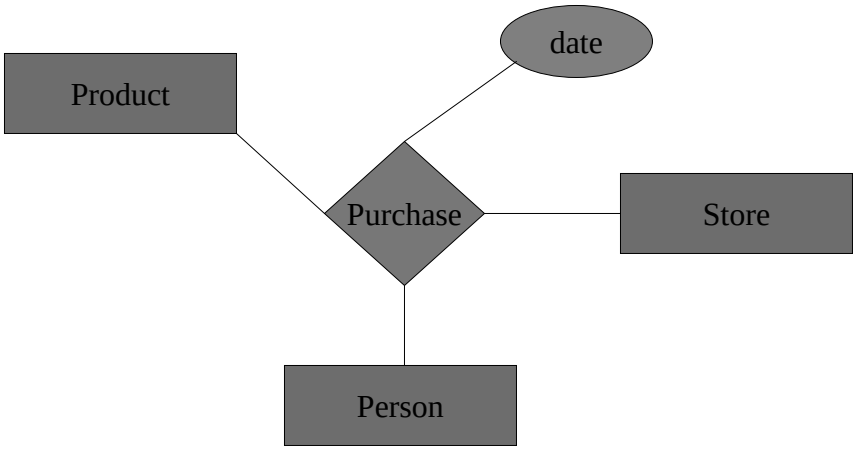


Roles in Relationships

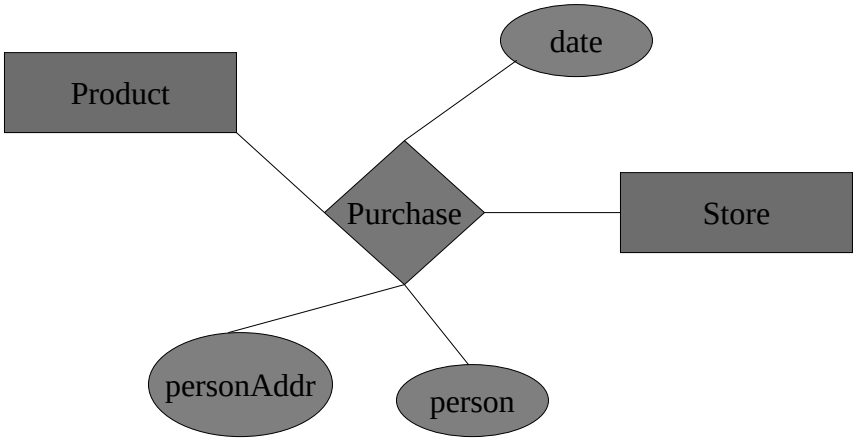
What if we need an entity set twice in one relationship?



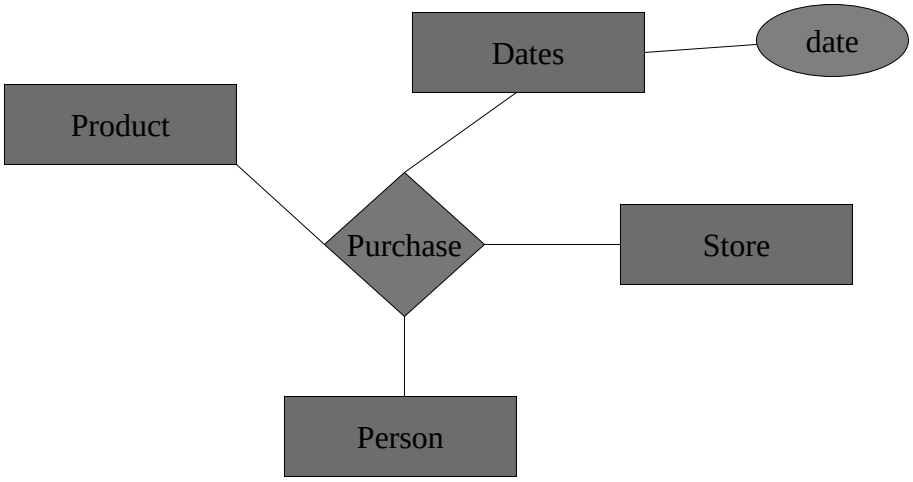
Attributes on Relationships



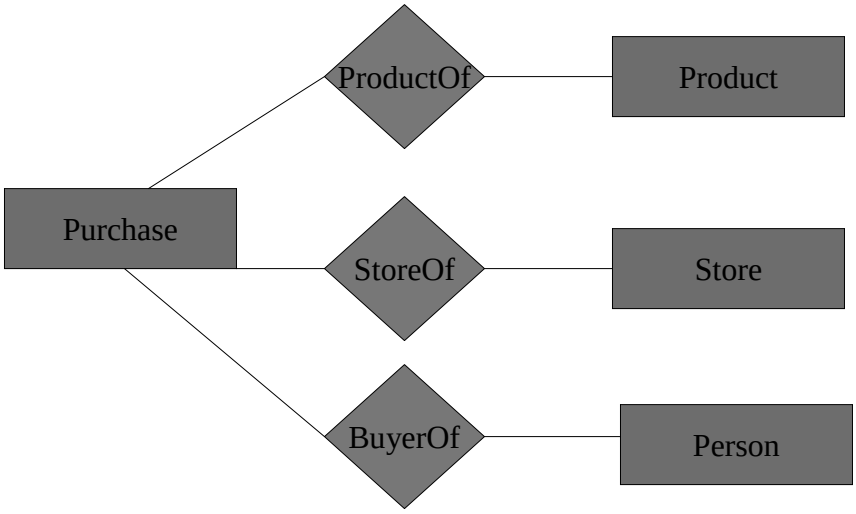
What's Wrong?



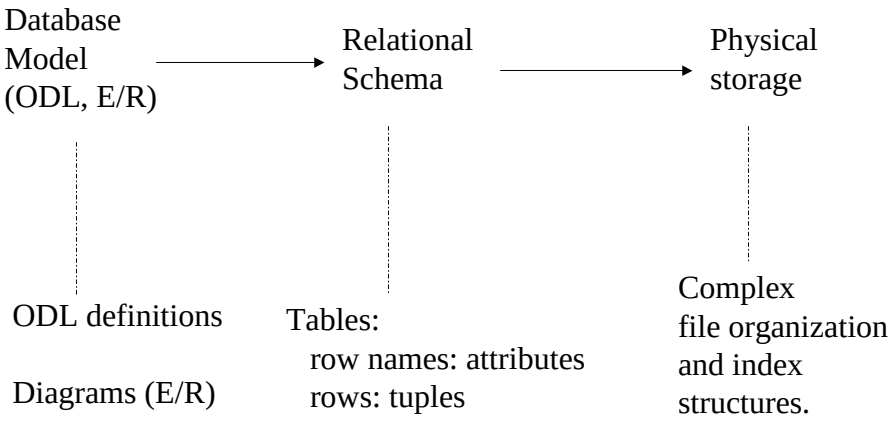
What's Wrong?



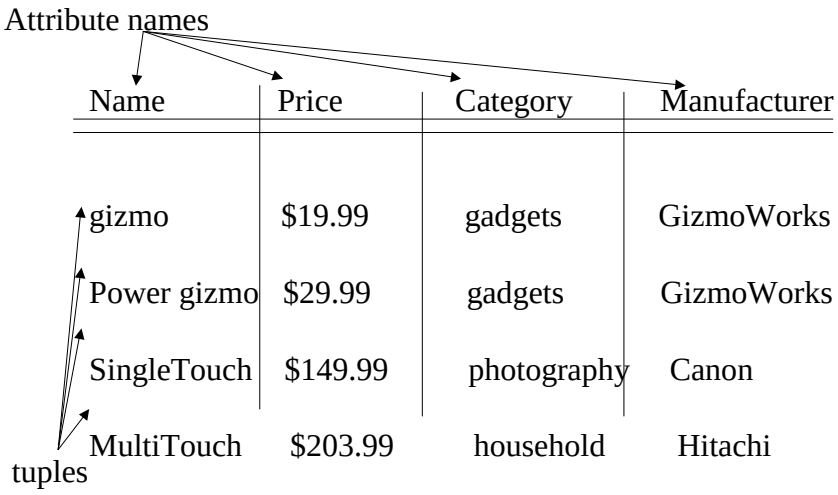
Do we really need 3-way relationships?



The Relational Data Model



Terminology



More Terminology

Every attribute has an atomic type.

Relation Schema: relation name + attribute names + attribute types

Relation instance: a set of tuples. Only one copy of any tuple!

Database Schema: a set of relation schemas.

Database instance: a relation instance for every relation in the schema.

More on Tuples

Formally, a mapping from attribute names to (correctly typed) values:

name → gizmo
price → $19.99
category → gadgets
manufacturer → GizmoWorks

Sometimes we refer to a tuple by itself: (note order of attributes)

(gizmo, $19.99, gadgets, GizmoWorks) or

Product (gizmo, $19.99, gadgets, GizmoWorks).

Updates

The database maintains a current database state.

Updates to the data:

1) add a tuple
2) delete a tuple
3) modify an attribute in a tuple

Updates to the data happen very frequently.

Updates to the schema: relatively rare. Rather painful.

From ODL to Relational Schema

Start simple: a class definition has only single valued attributes

```
Interface product{
        float price;
        string name;
        Enum {telephony, gadgets, books}  category}
```

Every attribute becomes a relation attribute:

Name	Price	Category
Gizmo	$19.99	gadgets

Adding Non atomic Attributes

Price is a record: {string currency, float amount}

Name	Currency	Amount	Category
Gizmo	US$	19.99	gadgets
Power Gizmo	US$	29.99	gadgets

Set Attributes

One option: have a tuple for every value in the set:

Name	SSN	Phone Number
Fred	123-321-99	(201) 555-1234
Fred	123-321-99	(206) 572-4312
Joe	909-438-44	(908) 464-0028
Joe	909-438-44	(212) 555-4000

Disadvantages?

Modeling Collection Types

The problem becomes even more significant if a class has several attributes that are set types?

Question: how bad is the redundancy for *n* set type attributes, each with possibly up to *m* values?

Questions:

How can we model bags?

Lists?

Fixed length arrays?

Modeling Relationships

```
Interface Product {
        attribute  string name;
        attribute  float   price;
        relationship  <Company>  madeBy;
}

Interface  Company {
         attribute   string  name;
         attribute   float stock-price;
         attribute   string  address;
}
```

How do we incorporate the relationship madeBy into the schema?

Option #1

Name	Price	made-by-name	made-by-stock-price	made-by-address
Gizmo	$19.99	gizmoWorks	0.0001$	Montezuma

What's wrong?

Hint

```
Interface Product {
        attribute  string name;
        attribute  float   price;
        relationship  <Company>  madeBy;
}

Interface  Company {
         attribute   string  name;
         attribute   float stock-price;
         attribute   string  address;
         relationship  set <Product> makes;
}
```

Better Solution

Product relation: (assume: name is a key for company)

Name	Price	made-by-name
Gizmo	$19.99	GizmoWorks

Company relation:

Name	Stock Price	Address
GizmoWorks	$0.00001	Montezuma

From E/R Diagrams to Relational Schema

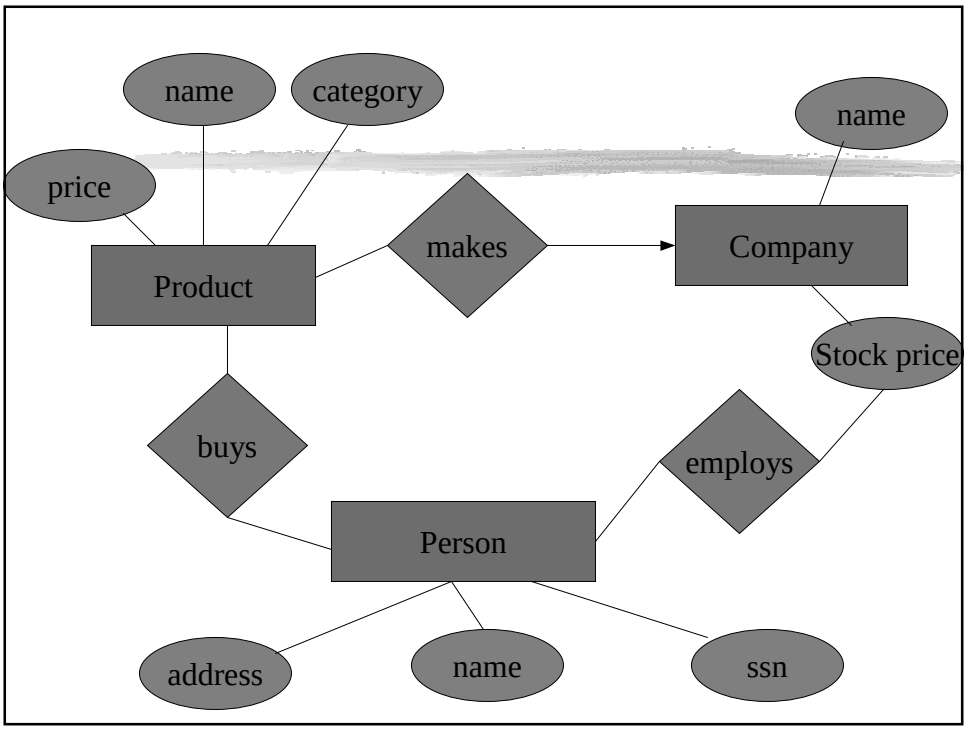
Easier than ODL (using a liberal interpretation of the word “easy”)

- relationships are already independent entities
- only atomic types exist in the E/R model.

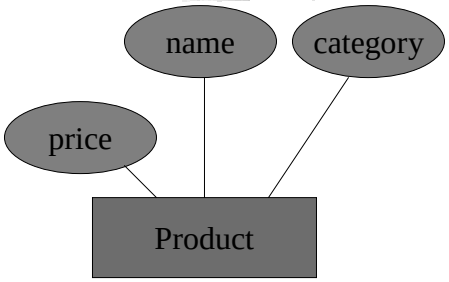
Entity sets → relations

Relationships → relations

Special care for weak entity sets.



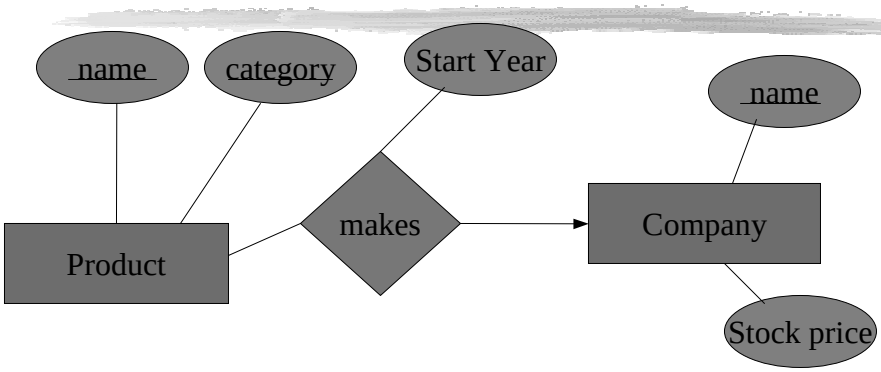
Entity Sets to Relations



Product:

Name	Category	Price
gizmo	gadgets	$19.99

Relationships to Relations



Relation MAKES (watch out for attribute name conflicts)

Product-name	Product-Category	Company-name	Starting-year
gizmo	gadgets	gizmoWorks	1963

Problems in Designing Schema

Name	SSN	Phone Number
Fred	123-321-99	(201) 555-1234
Fred	123-321-99	(206) 572-4312
Joe	909-438-44	(908) 464-0028
Joe	909-438-44	(212) 555-4000

Problems:
- redundancy
- update anomalies
- deletion anomalies

Relation Decomposition

Break the relation into two relations:

Name	SSN
Fred	123-321-99
Joe	909-438-44

Name	Phone Number
Fred	(201) 555-1234
Fred	(206) 572-4312
Joe	(908) 464-0028
Joe	(212) 555-4000

Decompositions in General

Let *R* be a relation with attributes $A_1, A_2, \ldots A_n$

Create two relations *R1* and *R2* with attributes

$B_1, B_2, \ldots B_m$ $C_1, C_2, \ldots C_l$

Such that:

$B_1, B_2, \ldots B_m \cup C_1, C_2, \ldots C_l = A_1, A_2, \ldots A_n$

And

-- *R1* is the projection of *R* on $B_1, B_2, \ldots B_m$

-- *R2* is the projection of *R* on $C_1, C_2, \ldots C_l$

Boyce-Codd Normal Form

A simple condition for removing anomalies from relations:

A relation R is in BCNF if and only if:

Whenever there is a nontrivial dependency $A_1, A_2, \ldots A_n \rightarrow B$ for R , it is the case that $\{ A_1, A_2, \ldots A_n \}$ a super-key for R.

In English (though a bit vague):

Whenever a set of attributes of *R* is determining another attribute, should determine ***all*** the attributes of *R*.

Example

Name	SSN	Phone Number
Fred	123-321-99	(201) 555-1234
Fred	123-321-99	(206) 572-4312
Joe	909-438-44	(908) 464-0028
Joe	909-438-44	(212) 555-4000

What are the dependencies?

What are the keys?

Is it in BCNF?

And Now?

Name	SSN
Fred	123-321-99
Joe	909-438-44

Name	Phone Number
Fred	(201) 555-1234
Fred	(206) 572-4312
Joe	(908) 464-0028
Joe	(212) 555-4000

What About This?

Name	Price	Category
Gizmo	$19.99	gadgets

Question:

Find an example of a 2-attribute relation that is not in BCNF.

Let's Decompose!

Name	Address	Move-Date

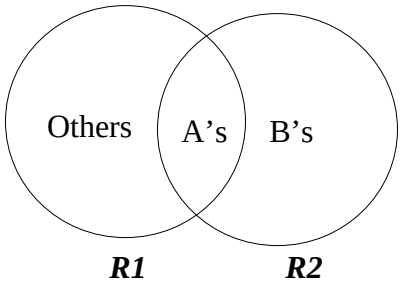
Name	Address

Name	Move-Date

More Careful Strategy

Find a dependency that violates the BCNF condition:

$$A_1, A_2, \ldots A_n \longrightarrow B_1, B_2, \ldots B_m$$



Example Decomposition

Name	Social-security-number	Age	Eye Color	Phone Number

Functional dependencies:

Name + Social-security-number $\longrightarrow$ Age, Eye Color

What if we also had an attribute Draft-worthy, and the FD:

Age $\longrightarrow$ Draft-worthy

Decomposition Based on BCNF is Necessarily Correct

Attributes A, B, C. FD: A $\longrightarrow$ C

Relations R1[A,B] R2[A,C]

Tuples in R1: (a,b)

Tuples in R2: (a,c), (a,d)

Tuples in the join of R1 and R2: (a,b,c), (a,b,d)

Can (a,b,d) be a bogus tuple?

Multivalued Dependencies

Name	SSN	Phone Number	Course
Fred	123-321-99	(206) 572-4312	CSE-444
Fred	123-321-99	(206) 572-4312	CSE-341
Fred	123-321-99	(206) 432-8954	CSE-444
Fred	123-321-99	(206) 432-8954	CSE-341

The multivalued dependencies are:

Name, SSN $\longrightarrow\!\!\!\!\!\rightarrow$ Phone Number
Name, SSN $\longrightarrow\!\!\!\!\!\rightarrow$ Course

4th Normal form: replace FD by MVD.

SQL Introduction

Standard language for querying and manipulating data

Structured **Q**uery **L**anguage

Many standards out there: SQL92, SQL2, SQL3.
Vendors support various subsets of these, but all of what we'll be talking about.

Basic form: (many many more bells and whistles in addition)

Select attributes
From relations (possibly multiple, joined)
Where conditions (selections)

Selections

```
SELECT   *
FROM     Company
WHERE    country="USA" AND stockPrice > 50
```

You can use:

- attribute names of the relation(s) used in the FROM.
- comparison operators: =, <>, <, >, <=, >=
- apply arithmetic operations: stockprice*2
- operations on strings (e.g., "||" for concatenation).
- lexicographic order on strings.
- pattern matching: s LIKE p
- special stuff for comparing dates and times.

Projections

Select only a subset of the attributes

```
SELECT   name, stock price
FROM     Company
WHERE    country="USA" AND stockPrice > 50
```

Rename the attributes in the resulting table

```
SELECT   name AS company, stockprice AS price
FROM     Company
WHERE    country="USA" AND stockPrice > 50
```

Ordering the Results

```
SELECT   name, stock price
FROM     Company
WHERE    country="USA" AND stockPrice > 50
ORDERBY  country, name
```

Ordering is ascending, unless you specify the DESC keyword.

Ties are broken by the second attribute on the ORDERBY list, etc.

Joins

```
SELECT   name, store
FROM     Person, Purchase
WHERE    name=buyer AND city="Seattle"
                    AND product="gizmo"
```

Product (name, price, category, maker)
Purchase (buyer, seller, store, product)
Company (name, stock price, country)
Person(name, phone number, city)

Disambiguating Attributes

Find names of people buying telephony products:

```
SELECT   Person.name
         FROM     Person, Purchase, Product
         WHERE        Person.name=buyer
                  AND  product=Product.name
                  AND  Product.category="telephony"
```

Product (name, price, category, maker)
Purchase (buyer, seller, store, product)
Person(name, phone number, city)

Tuple Variables

Find pairs of companies making products in the same category

```
SELECT  product1.maker, product2.maker
        FROM    Product AS product1, Product AS product2
        WHERE       product1.category=product2.category
                AND  product1.maker <> product2.maker
```

Product (name, price, category, maker)

Union, Intersection, Difference

```
(SELECT  name
 FROM    Person
 WHERE   City="Seattle")

   UNION

(SELECT  name
 FROM    Person, Purchase
 WHERE   buyer = name AND store = "The Bon")
```

Similarly, you can use INTERSECT and EXCEPT.
You must have the same attribute names (otherwise: rename).

Subqueries

```
SELECT Purchase.product
FROM     Purchase
WHERE  buyer =
            (SELECT  name
              FROM     Person
              WHERE   social-security-number = “123 - 45 - 6789”);
```

In this case, the subquery returns one value.

If it returns more, it’s a run-time error.

Subqueries Returning Relations

Find companies who manufacture products bought by Joe Blow.

```
SELECT  Company.name
FROM      Company, Product
WHERE   Company.name=maker
          AND  Product.name  IN
                  (SELECT product
                    FROM   Purchase
                    WHERE  buyer = “Joe Blow”);
```

You can also use: s > ALL R
s > ANY R
EXISTS R

Conditions on Tuples

```
SELECT  Company.name
FROM      Company, Product
WHERE   Company.name=maker
        AND  (Product.name,price)  IN
                (SELECT product, price)
                  FROM   Purchase
                  WHERE  buyer = "Joe Blow");
```

Correlated Queries

Find movies whose title appears more than once.

```
SELECT  title
FROM   Movie AS Old
WHERE  year < ANY
                (SELECT  year
                  FROM    Movie
                  WHERE  title =  Old.title);
```

Movie (title, year, director, length)
Movie titles are not unique (titles may reappear in a later year).

Note scope of variables

Removing Duplicates

```
SELECT  DISTINCT Company.name
FROM      Company, Product
WHERE   Company.name=maker
          AND  (Product.name,price)  IN
                  (SELECT product, price)
                    FROM   Purchase
                    WHERE  buyer = “Joe Blow”);
```

Conserving Duplicates

The UNION, INTERSECTION and EXCEPT operators operate as sets, not bags.

```
(SELECT  name
 FROM      Person
 WHERE    City=“Seattle”)

    UNION  ALL

(SELECT  name
 FROM      Person, Purchase
 WHERE   buyer=name AND store=“The Bon”)
```

Aggregation

```
SELECT  Sum(price)
FROM    Product
WHERE   manufacturer="Toyota"
```

SQL supports several aggregation operations:

SUM, MIN, MAX, AVG, COUNT

Except COUNT, all aggregations apply to a single attribute

```
SELECT  Count(*)
FROM    Purchase
```

Grouping and Aggregation

Usually, we want aggregations on certain parts of the relation.

Find how much we sold of every product

```
SELECT      product, Sum(price)
FROM        Product,  Purchase
WHERE       Product.name = Purchase.product
GROUPBY     Product.name
```

1. Compute the relation (I.e., the FROM and WHERE).
2. Group by the attributes in the GROUPBY
3. Select one tuple for every group (and apply aggregation)

SELECT can have (1) grouped attributes or (2) aggregates.

HAVING Clause

Same query, except that we consider only products that had at least 100 buyers.

```
SELECT    product, Sum(price)
FROM      Product,  Purchase
WHERE     Product.name = Purchase.product
GROUPBY   Product.name
HAVING    Count(buyer) > 100
```

HAVING clause contains conditions on aggregates.

Modifying the Database

We have 3 kinds of modifications: insertion, deletion, update.

Insertion: general form --

INSERT INTO R(A1,…., An) VALUES (v1,…., vn)

Insert a new purchase to the database:

```
INSERT   INTO   Purchase(buyer, seller, product, store)
         VALUES  (Joe, Fred, wakeup-clock-espresso-machine,
                  "The Sharper Image")
```

If we don't provide all the attributes of R, they will be filled with NULL.

We can drop the attribute names if we're providing all of them in order.

More Interesting Insertions

```
INSERT   INTO   PRODUCT(name)

    SELECT  DISTINCT  product
    FROM  Purchase
    WHERE   product  NOT IN
                (SELECT  name
                  FROM     Product)
```

The query replaces the VALUES keyword.

Note the order of querying and inserting.

Deletions

```
DELETE    FROM    PURCHASE

WHERE    seller = "Joe"    AND
             product = "Brooklyn Bridge"
```

Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.

Updates

```
UPDATE   PRODUCT
SET    price = price/2
WHERE  Product.name  IN
            (SELECT product
              FROM    Sales
              WHERE  Date = today);
```

Defining Views

Views are relations, except that they are not physically stored.

They are used mostly in order to simplify complex queries and to define conceptually different views of the database to different classes of users.

View: purchases of telephony products:

```
CREATE VIEW  telephony-purchases AS
  SELECT product, buyer, seller, store
  FROM  Purchase, Product
  WHERE  Purchase.product = Product.name
         AND  Product.category = "telephony"
```

A Different View

CREATE VIEW Seattle-view AS

```
SELECT  buyer, seller, product, store
FROM    Person, Purchase
WHERE   Person.city = "Seattle"    AND
        Person.name = Purchase.buyer
```

We can later use the views:

```
SELECT   name, store
FROM     Seattle-view, Product
WHERE    Seattle-view.product = Product.name  AND
         Product.category = "shoes"
```

What's really happening when we query a view??