CSE544
SQL

Wednesday, March 31, 2004

Administrivia

- Sign up for the 544 mailing list!
- Assignment 1 is released. The deadline for first part is 7th April.

SQL Introduction

Standard language for querying and manipulating data

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

Many standards out there:
- ANSI SQL
- SQL92 (a.k.a. SQL2)
- SQL99 (a.k.a. SQL3)
- Vendors support various subsets of these
- What we discuss is common to all of them

SQL

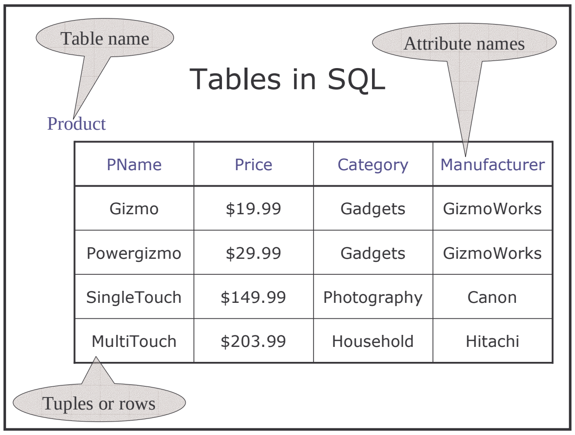
- Data Definition Language (DDL)
  - Create/alter/delete tables and their attributes
  - Following lectures...
- Data Manipulation Language (DML)
  - Query one or more tables – discussed next !
  - Insert/delete/modify tuples in tables
- Transact-SQL
  - Idea: package a sequence of SQL statements à server
  - Won’t discuss in class

Data in SQL

1. Atomic types, a.k.a. data types
2. Tables built from atomic types

Data Types in SQL

- Characters:
  - CHAR(20)   -- fixed length
  - VARCHAR(40)   -- variable length
- Numbers:
  - BIGINT, INT, SMALLINT, TINYINT
  - REAL, FLOAT   -- differ in precision
  - MONEY
- Times and dates:
  - DATE
  - DATETIME   -- SQL Server
- Others... All are simple



Tables in SQL

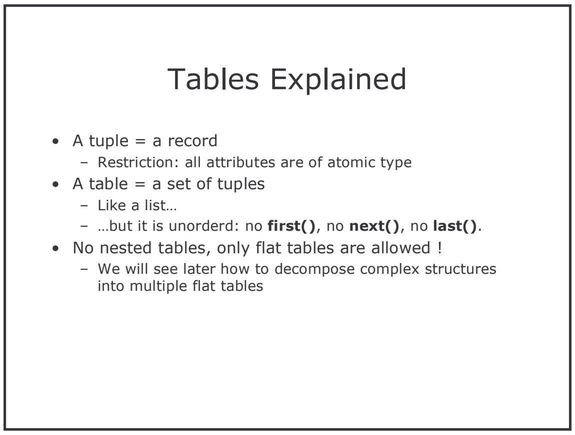
Table name

Attribute names

Product

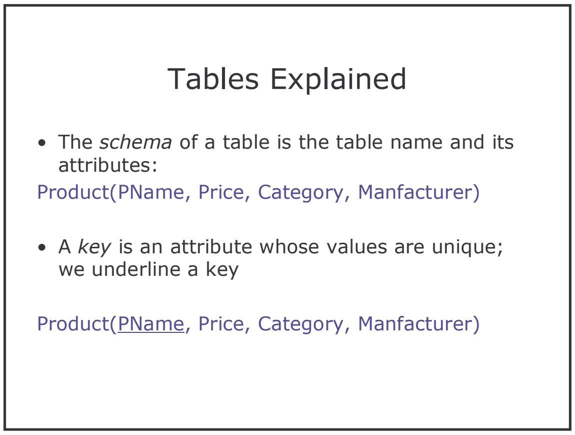
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Tuples or rows



Tables Explained

- A tuple = a record
  - Restriction: all attributes are of atomic type
- A table = a set of tuples
  - Like a list…
  - …but it is unorderd: no **first()**, no **next()**, no **last()**.
- No nested tables, only flat tables are allowed !
  - We will see later how to decompose complex structures into multiple flat tables



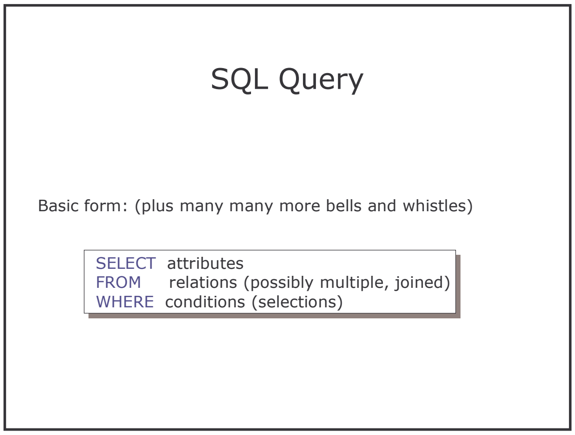
Tables Explained

- The *schema* of a table is the table name and its attributes:

Product(PName, Price, Category, Manfacturer)

- A *key* is an attribute whose values are unique; we underline a key

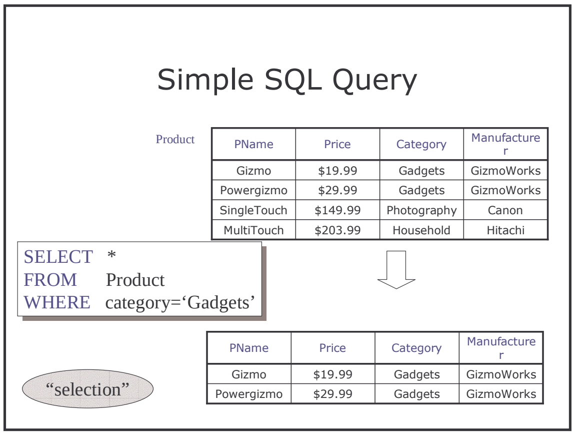
Product(<u>PName</u>, Price, Category, Manfacturer)



SQL Query

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

```
SELECT  attributes
FROM     relations (possibly multiple, joined)
WHERE  conditions (selections)
```



Simple SQL Query

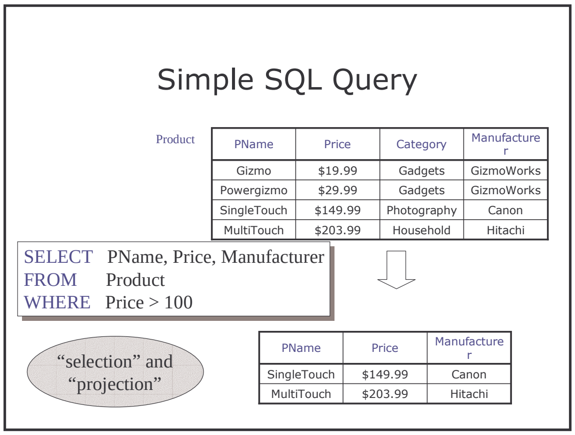
Product

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   *
FROM     Product
WHERE    category='Gadgets'
```

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

"selection"



Simple SQL Query

Product

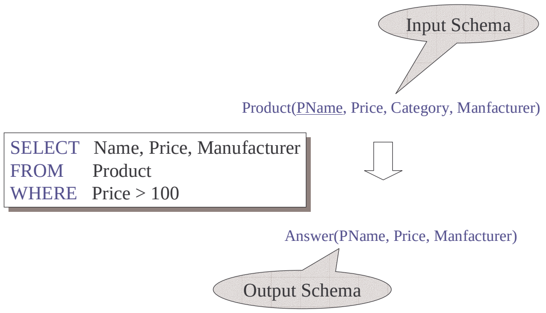
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

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

PName	Price	Manufacturer
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

"selection" and "projection"

A Notation for SQL Queries


Input Schema
Product(PName, Price, Category, Manfacturer)
Answer(PName, Price, Manfacturer)
Output Schema

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

Selections

What goes in the WHERE clause:

- x = y, x < y, x <= y, etc
  - For number, they have the usual meanings
  - For CHAR and VARCHAR: lexicographic ordering
    - Expected conversion between CHAR and VARCHAR
  - For dates and times, what you expect...
- Pattern matching on strings: s LIKE p (next)

The **LIKE** operator

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

Product(Name, Price, Category, Manufacturer)
Find all products whose name mentions ‘gizmo’:

```
SELECT   *
FROM     Products
WHERE    PName LIKE ‘%gizmo%’
```

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
```

Ordering is ascending, unless you specify the DESC keyword.

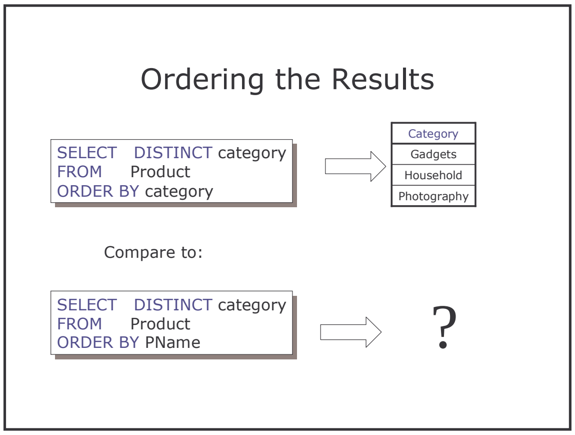
Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering the Results

```
SELECT   Category
FROM     Product
ORDER BY  PName
```

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

⇨ ?



Ordering the Results

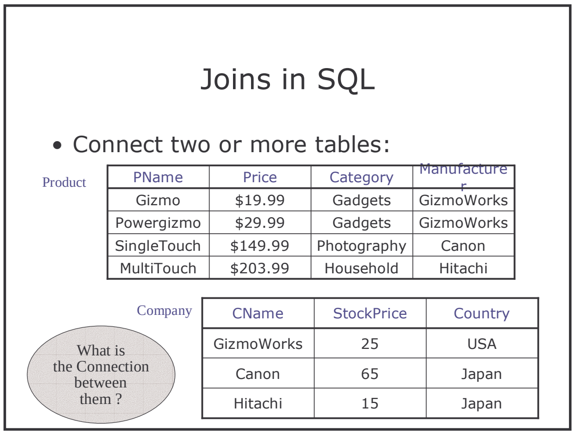
```
SELECT   DISTINCT category
FROM     Product
ORDER BY category
```

Category
Gadgets
Household
Photography

Compare to:

```
SELECT   DISTINCT category
FROM     Product
ORDER BY PName
```

?



Joins in SQL

- Connect two or more tables:

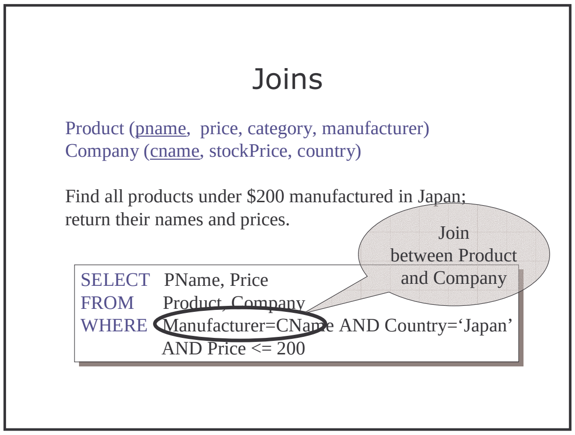
Product

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

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is the Connection between them ?



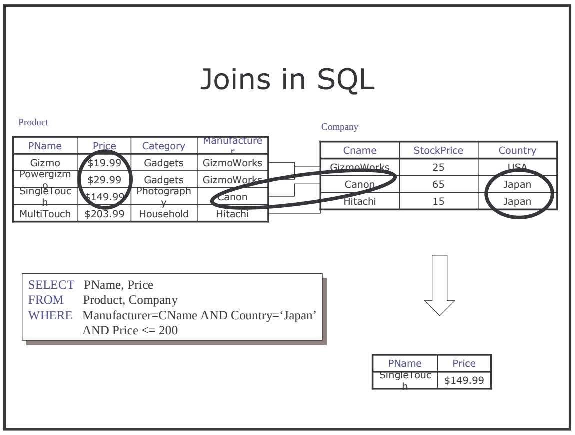
Joins

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

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

```
SELECT   PName, Price
FROM     Product, Company
WHERE    Manufacturer=CName AND Country=‘Japan’
         AND Price <= 200
```

Join between Product and Company



Joins in SQL

Product

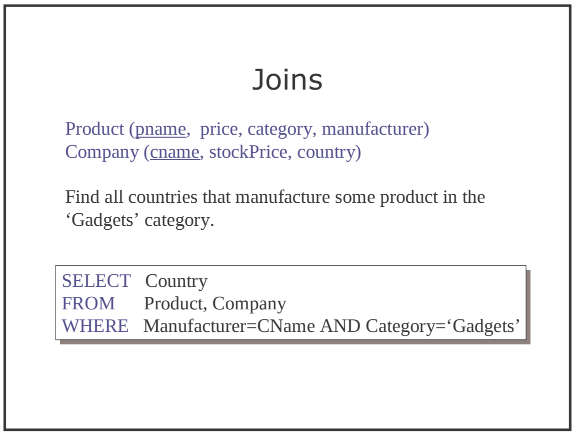
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT   PName, Price
FROM     Product, Company
WHERE    Manufacturer=CName AND Country=‘Japan’
         AND Price <= 200
```

PName	Price
SingleTouch	$149.99

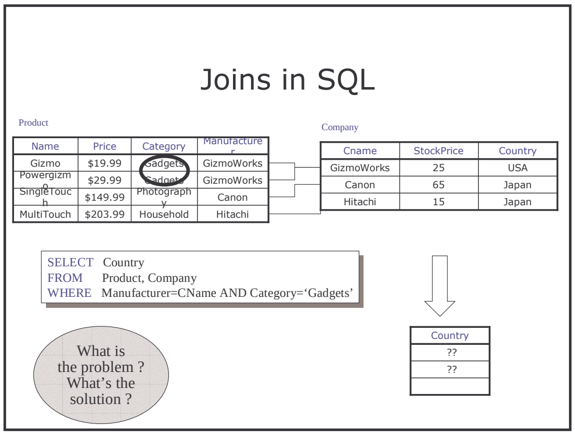


Joins

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

Find all countries that manufacture some product in the ‘Gadgets’ category.

```
SELECT   Country
FROM     Product, Company
WHERE    Manufacturer=CName AND Category=‘Gadgets’
```



Joins in SQL

Product

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

Company

Cname	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

```
SELECT   Country
FROM     Product, Company
WHERE    Manufacturer=CName AND Category=‘Gadgets’
```

Country
??
??

What is the problem ? What’s the solution ?

Joins

Product (pname, price, category, manufacturer)
Purchase (buyer, seller, store, product)
Person(persname, phoneNumber, city)

Find names of people living in Seattle that bought some product in the 'Gadgets' category, and the names of the stores they bought such product from

```
SELECT   DISTINCT persname, store
FROM     Person, Purchase, Product
WHERE    persname=buyer AND product = pname AND
         city='Seattle'  AND category='Gadgets'
```

Disambiguating Attributes

- Sometimes two relations have the same attr:
  Person(pname, address, worksfor)
  Company(cname, address)

```
SELECT   DISTINCT pname, address
FROM     Person, Company
WHERE    worksfor = cname
```

Which address ?

⇩

```
SELECT   DISTINCT Person.pname, Company.address
FROM     Person, Company
WHERE    Person.worksfor = Company.cname
```

Tuple Variables in SQL

Purchase (buyer, seller, store, product)

Find all stores that sold at least one product that was sold at 'BestBuy':

```
SELECT DISTINCT  x.store
FROM     Purchase AS x, Purchase AS y
WHERE    x.product = y.product AND y.store = 'BestBuy'
```

Tuple Variables

General rule:
tuple variables introduced automatically by the system:

Product (name, price, category, manufacturer)

```
SELECT  name
FROM    Product
WHERE   price > 100
```

Becomes:

```
SELECT Product.name
FROM    Product AS Product
WHERE Product.price > 100
```

Doesn't work when Product occurs more than once:
In that case the user needs to define variables explicitely.

Meaning (Semantics) of SQL Queries

```
SELECT a1, a2, …, ak
FROM    R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

1. Nested loops:

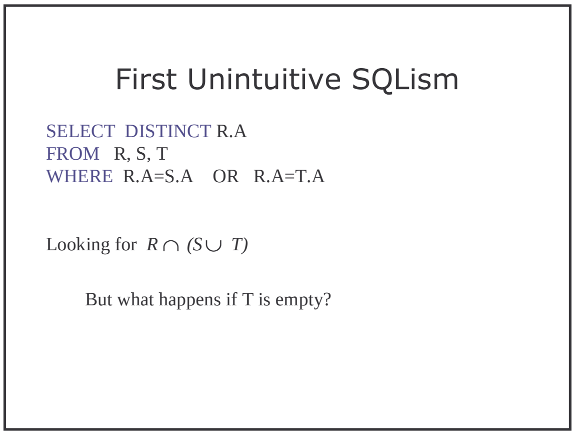
```
Answer = {}
for x1 in R1 do
    for x2 in R2 do
        …..
            for xn in Rn do
                if Conditions
                    then Answer = Answer ∪ {(a1,…,ak)}
return Answer
```

Meaning (Semantics) of SQL Queries

```
SELECT a1, a2, …, ak
FROM    R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions
```

2. Parallel assignment

```
Answer = {}
for all assignments x1 in R1, …, xn in Rn do
    if Conditions then Answer = Answer ∪ {(a1,…,ak)}
return Answer
```

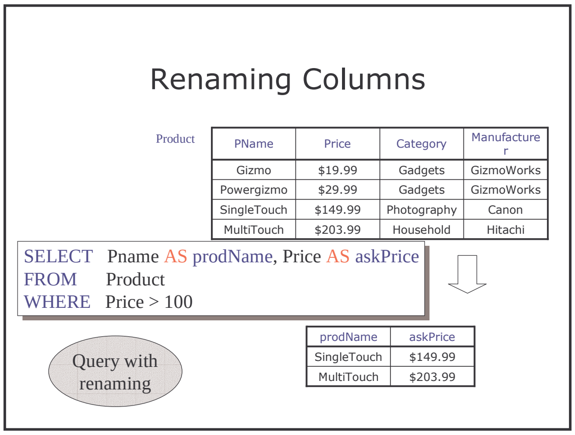


First Unintuitive SQLism

```
SELECT  DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A    OR   R.A=T.A
```

Looking for $R \cap (S \cup T)$

But what happens if T is empty?



Renaming Columns

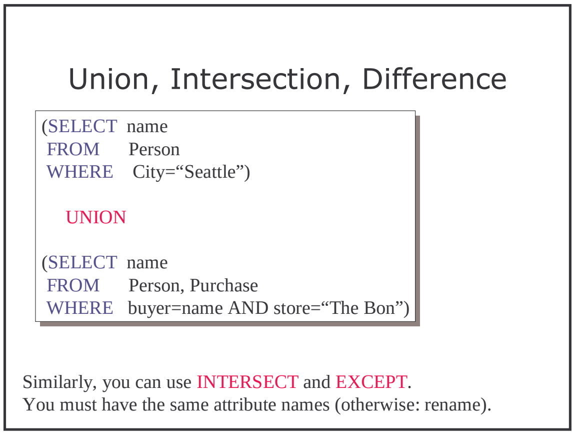
Product

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 AS prodName, Price AS askPrice
FROM     Product
WHERE    Price > 100
```

prodName	askPrice
SingleTouch	$149.99
MultiTouch	$203.99

Query with renaming



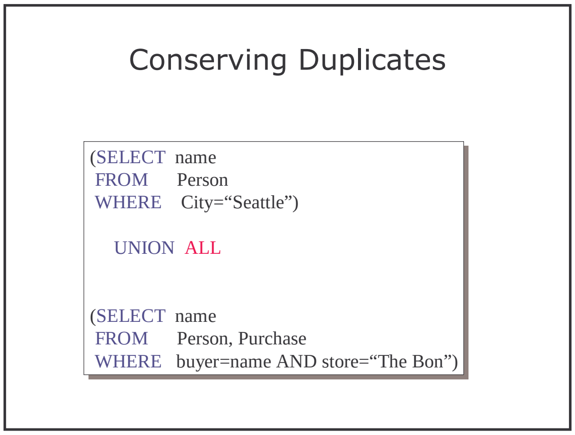
Union, Intersection, Difference

```
(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).

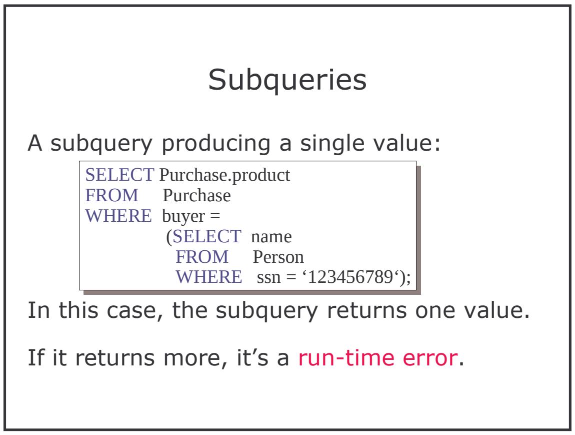


Conserving Duplicates

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

    UNION  ALL


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



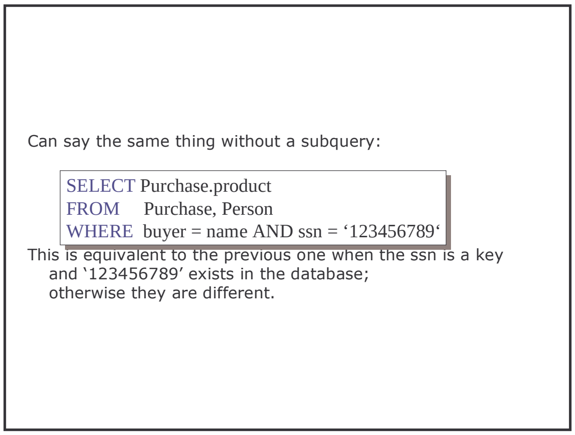
Subqueries

A subquery producing a single value:

```
SELECT Purchase.product
FROM     Purchase
WHERE  buyer =
           (SELECT  name
             FROM     Person
             WHERE   ssn = '123456789');
```

In this case, the subquery returns one value.

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



Can say the same thing without a subquery:

```
SELECT Purchase.product
FROM     Purchase, Person
WHERE  buyer = name AND ssn = '123456789'
```

This is equivalent to the previous one when the ssn is a key
and '123456789' exists in the database;
otherwise they are different.

Subqueries Returning Relations

Find companies that manufacture products bought by Joe Blow.

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

Here the subquery returns a set of values: no more runtime errors.

Subqueries Returning Relations

Equivalent to:

```
SELECT  Company.name
FROM     Company, Product, Purchase
WHERE   Company.name= Product.maker
           AND  Product.name  = Purchase.product
           AND  Purchase.buyer = ‘Joe Blow’
```

Is this query equivalent to the previous one ?

Beware of duplicates !

Removing Duplicates

```
SELECT DISTINCT Company.name
FROM     Company, Product
WHERE   Company.name= Product.maker
           AND  Product.name  IN
                   (SELECT Purchase.product
                    FROM    Purchase
                    WHERE Purchase.buyer = ‘Joe Blow’)
```

```
SELECT DISTINCT Company.name
FROM     Company, Product, Purchase
WHERE   Company.name= Product.maker
           AND  Product.name  = Purchase.product
           AND  Purchase.buyer = ‘Joe Blow’
```

Now they are equivalent

Subqueries Returning Relations

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

Product (pname, price, category, maker)
Find products that are more expensive than all those produced By “Gizmo-Works”

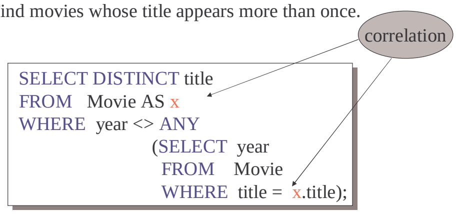
```
SELECT  name
FROM     Product
WHERE  price >  ALL (SELECT price
                              FROM     Purchase
                              WHERE  maker=‘Gizmo-Works’)
```

Question for Database Fans and their Friends

- Can we express this query as a single SELECT-FROM-WHERE query, without subqueries ?
- Hint: show that all SFW queries are monotone (figure out what this means). A query with **ALL** is not monotone

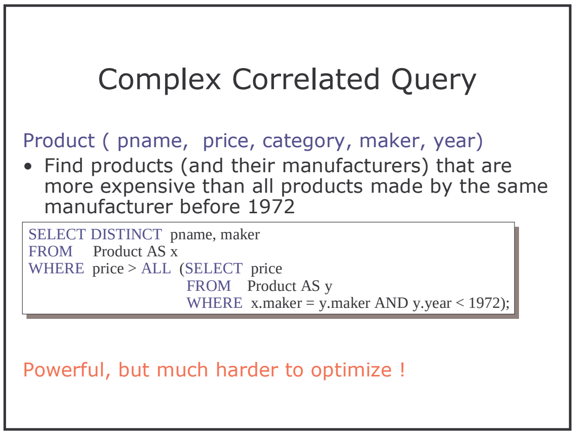
Correlated Queries

Movie (title, year, director, length)
Find movies whose title appears more than once.



```
SELECT DISTINCT title
FROM   Movie AS  x
WHERE  year <> ANY
                   (SELECT  year
                    FROM    Movie
                    WHERE  title =  x.title);
```

Note (1) scope of variables (2) this can still be expressed as single SFW



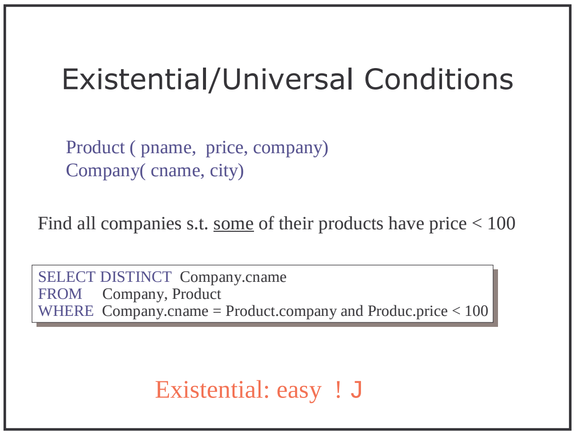
Complex Correlated Query

Product (pname, price, category, maker, year)

- Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

```
SELECT DISTINCT  pname, maker
FROM     Product AS x
WHERE  price > ALL  (SELECT  price
                              FROM    Product AS y
                              WHERE  x.maker = y.maker AND y.year < 1972);
```

Powerful, but much harder to optimize !



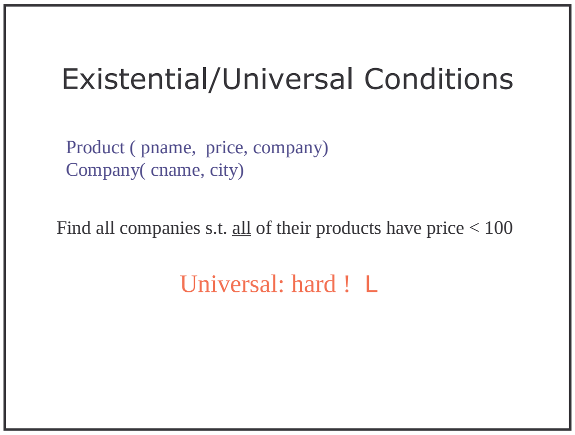
Existential/Universal Conditions

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

Find all companies s.t. some of their products have price < 100

```
SELECT DISTINCT  Company.cname
FROM     Company, Product
WHERE  Company.cname = Product.company and Produc.price < 100
```

Existential: easy ! J

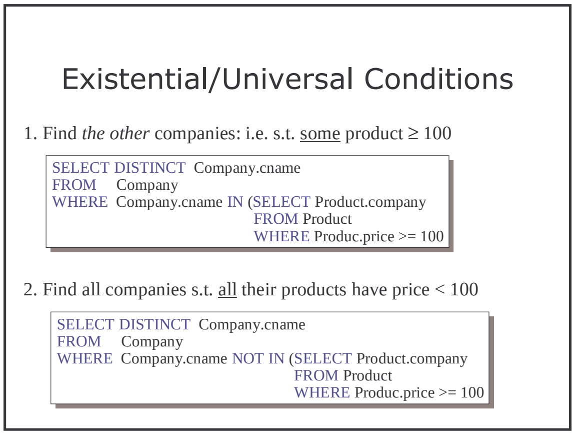


Existential/Universal Conditions

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

Find all companies s.t. all of their products have price < 100

Universal: hard ! L



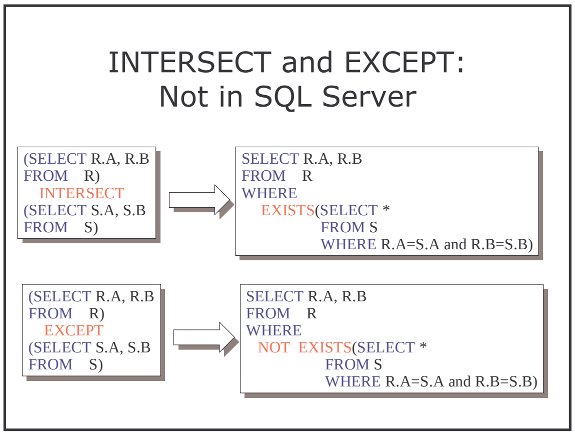
Existential/Universal Conditions

1. Find *the other* companies: i.e. s.t. some product ≥ 100

```
SELECT DISTINCT  Company.cname
FROM     Company
WHERE  Company.cname IN (SELECT Product.company
                                          FROM Product
                                          WHERE Produc.price >= 100
```

2. Find all companies s.t. all their products have price < 100

```
SELECT DISTINCT  Company.cname
FROM     Company
WHERE  Company.cname NOT IN (SELECT Product.company
                                                  FROM Product
                                                  WHERE Produc.price >= 100
```



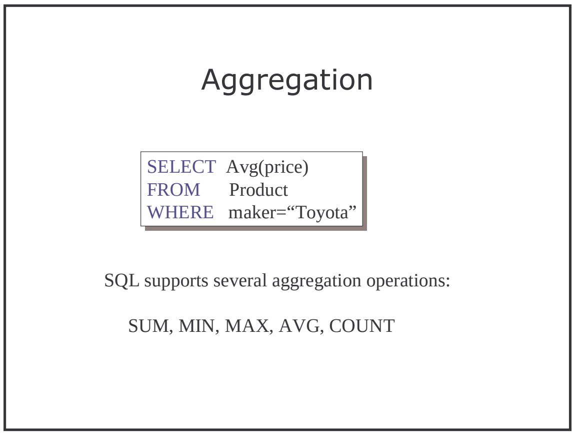
INTERSECT and EXCEPT: Not in SQL Server

```
(SELECT R.A, R.B
FROM    R)
   INTERSECT
(SELECT S.A, S.B
FROM    S)
```

```
SELECT R.A, R.B
FROM    R
WHERE
    EXISTS(SELECT *
              FROM S
              WHERE R.A=S.A and R.B=S.B)
```

```
(SELECT R.A, R.B
FROM    R)
   EXCEPT
(SELECT S.A, S.B
FROM    S)
```

```
SELECT R.A, R.B
FROM    R
WHERE
  NOT  EXISTS(SELECT *
              FROM S
              WHERE R.A=S.A and R.B=S.B)
```



Aggregation

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

SQL supports several aggregation operations:

SUM, MIN, MAX, AVG, COUNT

Aggregation: Count

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

Except COUNT, all aggregations apply to a single attribute

Aggregation: Count

COUNT   applies to duplicates, unless otherwise stated:

```
SELECT  Count(category)
FROM     Product
WHERE   year > 1995
```

same as Count(*)

Better:

```
SELECT  Count(DISTINCT category)
FROM     Product
WHERE   year > 1995
```

Simple Aggregation

Purchase(product, date, price, quantity)

Example 1: find total sales for the entire database

```
SELECT  Sum(price * quantity)
FROM      Purchase
```

Example 1': find total sales of bagels

```
SELECT  Sum(price * quantity)
FROM      Purchase
WHERE   product = 'bagel'
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
Bagel	10/21	0.85	15
Banana	10/22	0.52	7
Banana	10/19	0.52	17
Bagel	10/20	0.85	20

Grouping and Aggregation

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

Purchase(product, date, price, quantity)

Example 2: **find total sales after 9/1 per product.**

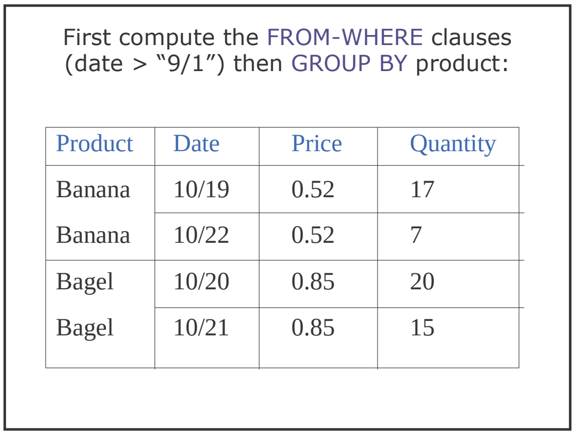
```
SELECT       product, Sum(price*quantity) AS TotalSales
FROM          Purchase
WHERE       date > "9/1"
GROUPBY  product
```

Let's see what this means…

Grouping and Aggregation

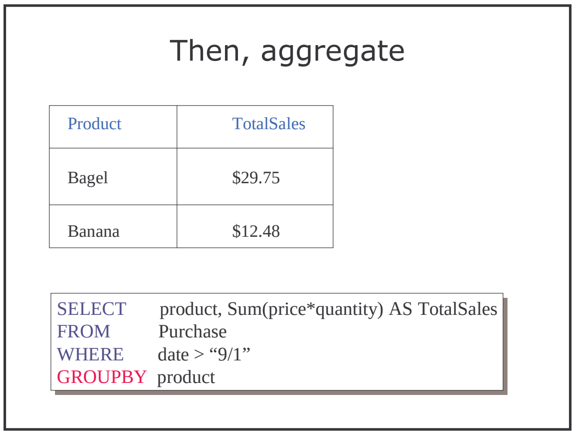
1. Compute the FROM and WHERE clauses.
2. Group by the attributes in the GROUPBY
3. Select one tuple for every group (and apply aggregation)

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



First compute the FROM-WHERE clauses (date > “9/1”) then GROUP BY product:

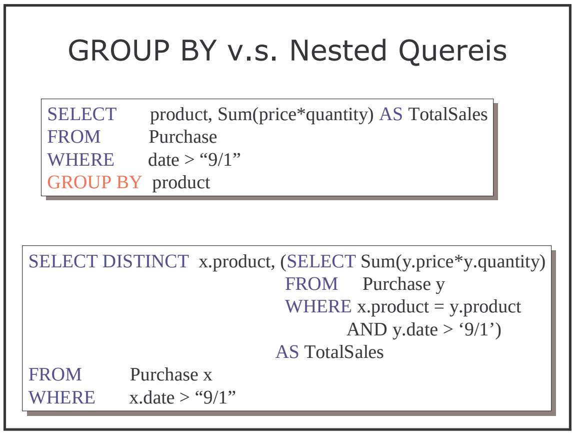
Product	Date	Price	Quantity
Banana	10/19	0.52	17
Banana	10/22	0.52	7
Bagel	10/20	0.85	20
Bagel	10/21	0.85	15



Then, aggregate

Product	TotalSales
Bagel	$29.75
Banana	$12.48

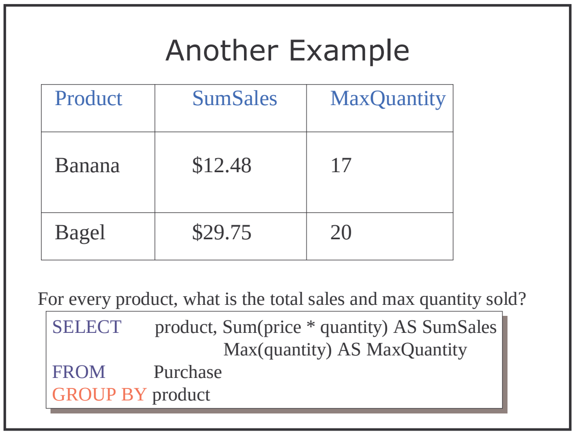
```
SELECT     product, Sum(price*quantity) AS TotalSales
FROM       Purchase
WHERE      date > “9/1”
GROUPBY  product
```



GROUP BY v.s. Nested Quereis

```
SELECT       product, Sum(price*quantity) AS TotalSales
FROM         Purchase
WHERE        date > “9/1”
GROUP BY  product
```

```
SELECT DISTINCT  x.product, (SELECT Sum(y.price*y.quantity)
                             FROM     Purchase y
                             WHERE x.product = y.product
                                   AND y.date > ‘9/1’)
                            AS TotalSales
FROM         Purchase x
WHERE        x.date > “9/1”
```

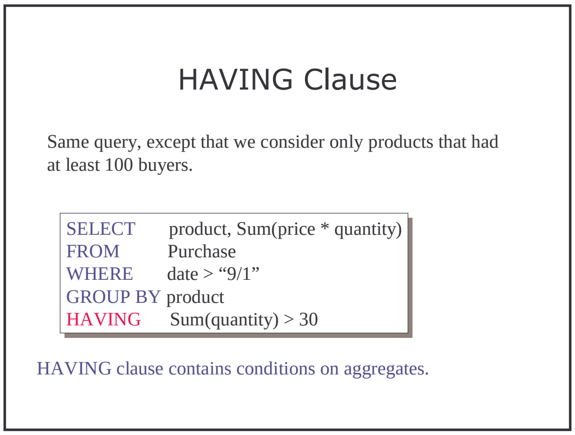


Another Example

Product	SumSales	MaxQuantity
Banana	$12.48	17
Bagel	$29.75	20

For every product, what is the total sales and max quantity sold?

```
SELECT       product, Sum(price * quantity) AS SumSales
                       Max(quantity) AS MaxQuantity
FROM         Purchase
GROUP BY product
```

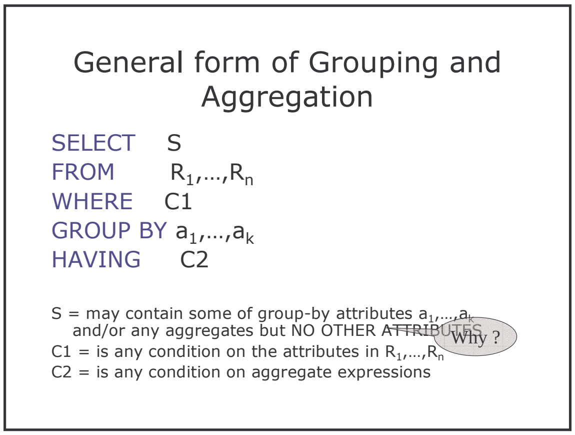


HAVING Clause

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

```
SELECT       product, Sum(price * quantity)
FROM         Purchase
WHERE        date > “9/1”
GROUP BY product
HAVING       Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.



General form of Grouping and Aggregation

SELECT S
FROM $R_1,\ldots,R_n$
WHERE C1
GROUP BY $a_1,\ldots,a_k$
HAVING C2

S = may contain some of group-by attributes $a_1,\ldots,a_k$ and/or any aggregates but NO OTHER ATTRIBUTES

Why ?

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

C2 = is any condition on aggregate expressions

General form of Grouping and Aggregation

SELECT S
FROM $R_1,…,R_n$
WHERE C1
GROUP BY $a_1,…,a_k$
HAVING C2

Evaluation steps:

1. Compute the FROM-WHERE part, obtain a table with all attributes in $R_1,…,R_n$
2. Group by the attributes $a_1,…,a_k$
3. Compute the aggregates in C2 and keep only groups satisfying C2
4. Compute aggregates in S and return the result

Examples of Queries with Aggregation

Web pages, and their authors:

Author(login,name)
Document(url, title)
Wrote(login,url)
Mentions(url,word)

- Find all authors who wrote at least 10 documents Author(login,name) Wrote(login,url)
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM          Author
WHERE         count(SELECT Wrote.url
                    FROM Wrote
                    WHERE Author.login=Wrote.login)
                  > 10
```

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT      Author.name
FROM        Author, Wrote
WHERE       Author.login=Wrote.login
GROUP BY    Author.login, Author.name
HAVING      count(wrote.url) > 10
```

No need for DISTINCT: automatically from GROUP BY

- Find all authors who have a vocabulary over 10000 words:

```
SELECT      Author.name
FROM        Author, Wrote, Mentions
WHERE       Author.login=Wrote.login AND Wrote.url=Mentions.url
GROUP BY    Author.name
HAVING      count(distinct Mentions.word) > 10000
```

Look carefully at the last two queries: you may be tempted to write them as a nested queries, but in SQL we write them best with GROUP BY

NULLS in SQL

- Whenever we don’t have a value, we can put a NULL
- Can mean many things:
  - Value does not exists
  - Value exists but is unknown
  - Value not applicable
  - Etc.
- The schema specifies for each attribute if can be null (*nullable* attribute) or not
- How does SQL cope with tables that have NULLs ?

Null Values

- If x= NULL then 4*(3-x)/7 is still NULL
- If x= NULL then x="Joe" is UNKNOWN
- In SQL there are three boolean values:

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM Person
WHERE  (age < 25) AND
            (height > 6 OR weight > 190)
```

JE

E.g.
age=20
heigth=NULL
weight=200

Null Values

Unexpected behavior:

```
SELECT *
FROM     Person
WHERE  age < 25  OR  age >= 25
```

Some Persons are not included !

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM     Person
WHERE  age < 25  OR  age >= 25 OR age IS NULL
```

Now it includes all Persons

Outerjoins

Product(name, category)
Purchase(prodName, store)

Display list of all products along with the stores where they were sold:

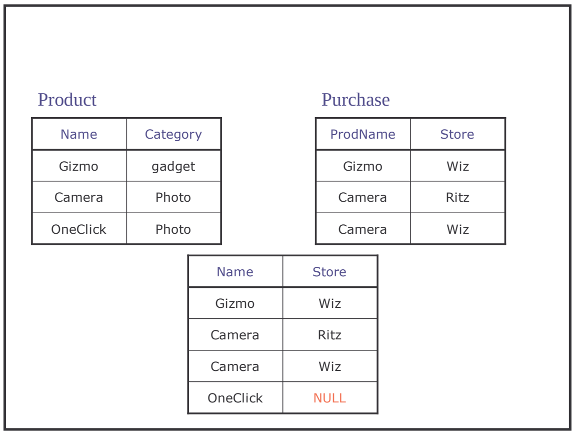
```
SELECT Product.name, Purchase.store
FROM     Product, Purchase
WHERE   Product.name = Purchase.prodName
```

But Products that never sold will be lost !

Outerjoins

Left outer joins in SQL:
Product(name, category)
Purchase(prodName, store)

```
SELECT Product.name, Purchase.store
FROM     Product LEFT OUTER JOIN Purchase ON
                   Product.name = Purchase.prodName
```



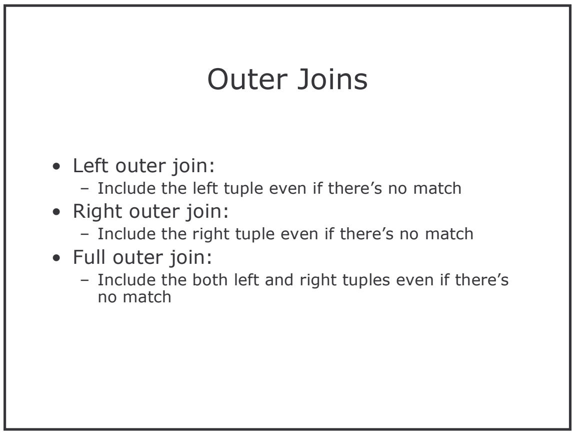
Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

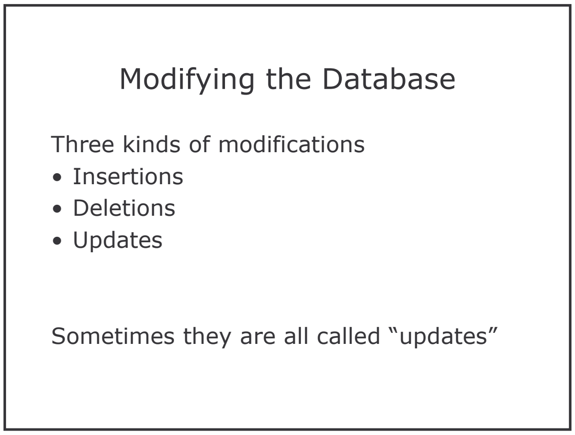
ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL



Outer Joins

- Left outer join:
  - Include the left tuple even if there's no match
- Right outer join:
  - Include the right tuple even if there's no match
- Full outer join:
  - Include the both left and right tuples even if there's no match

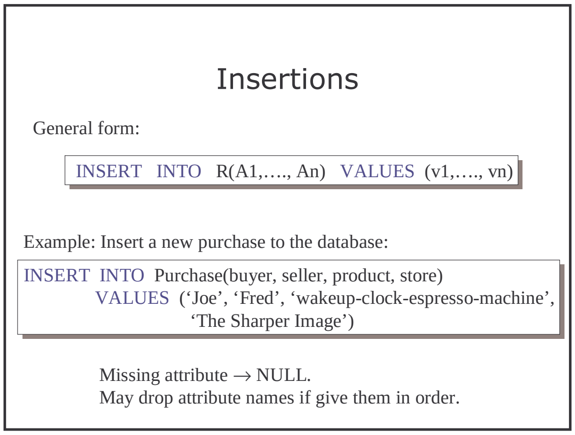


Modifying the Database

Three kinds of modifications

- Insertions
- Deletions
- Updates

Sometimes they are all called "updates"



Insertions

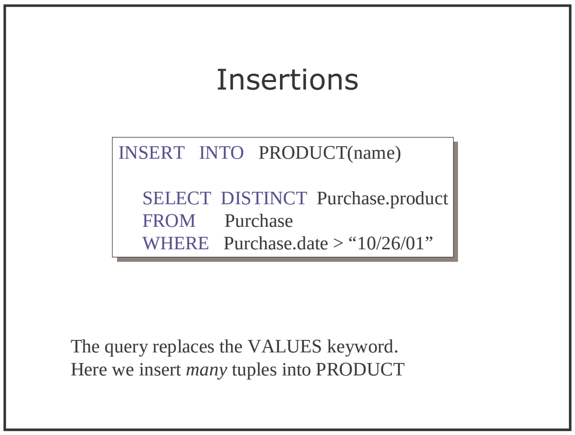
General form:

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

Example: Insert a new purchase to the database:

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

Missing attribute → NULL.
May drop attribute names if give them in order.

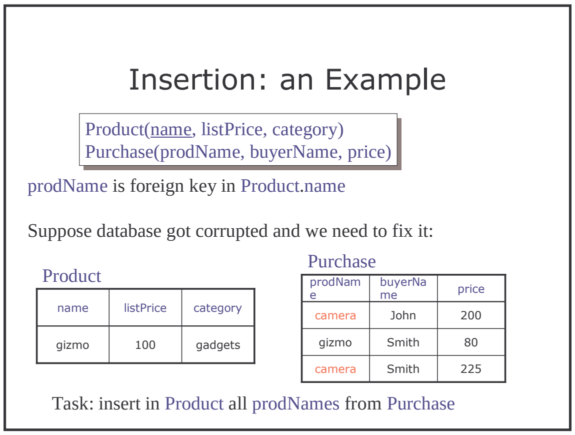


Insertions

```
INSERT  INTO  PRODUCT(name)

   SELECT  DISTINCT  Purchase.product
   FROM       Purchase
   WHERE   Purchase.date > "10/26/01"
```

The query replaces the VALUES keyword.
Here we insert *many* tuples into PRODUCT



Insertion: an Example

Product(name, listPrice, category)
Purchase(prodName, buyerName, price)

prodName is foreign key in Product.name

Suppose database got corrupted and we need to fix it:

Product

name	listPrice	category
gizmo	100	gadgets

Purchase

prodName	buyerName	price
camera	John	200
gizmo	Smith	80
camera	Smith	225

Task: insert in Product all prodNames from Purchase

Insertion: an Example

```
INSERT   INTO   Product(name)

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

name	listPrice	category
gizmo	100	Gadgets
camera	-	-

Insertion: an Example

```
INSERT   INTO   Product(name, listPrice)

 SELECT  DISTINCT  prodName, price
 FROM  Purchase
 WHERE   prodName  NOT IN (SELECT  name FROM  Product)
```

name	listPrice	category
gizmo	100	Gadgets
camera	200	-
camera ??	225 ??	-

← Depends on the implementation

Deletions

Example:

```
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

Example:

```
UPDATE   PRODUCT
SET    price = price/2
WHERE  Product.name  IN
              (SELECT product
               FROM    Purchase
               WHERE  Date =‘Oct, 25, 1999’);
```