Lecture 4: Updates, Views, Constraints and Other Fun Features

Monday, April 19th, 2004

Agenda

- Nulls and outerjoins
- Creating and updating schemas
- Views: updating and reusing them
- Constraints
- Programming with SQL
- Relational algebra

Null Values and Outerjoins

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

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

Same as:

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

But Products that never sold will be lost !

Null Values and 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

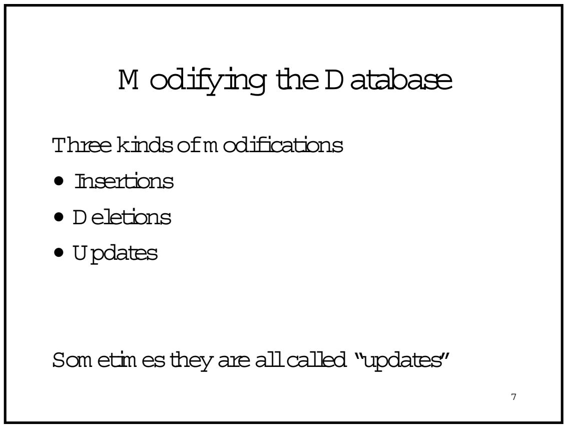
Purchase

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

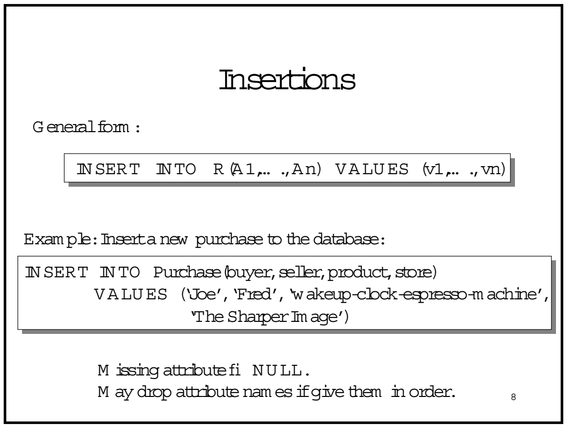


Modifying the Database

Three kinds of modifications

- Insertions
- Deletions
- Updates

Sometimes they are all called “updates”



Insertions

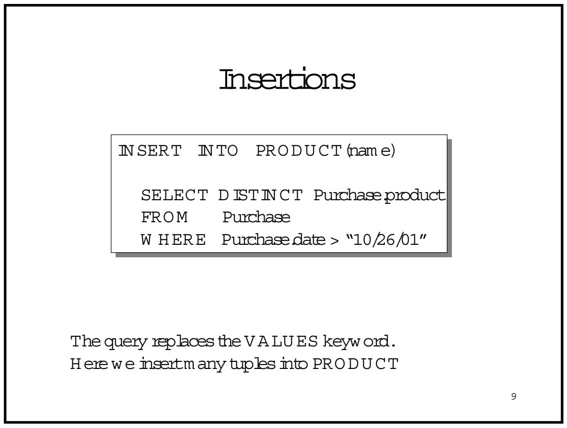
General form:

```
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')
```

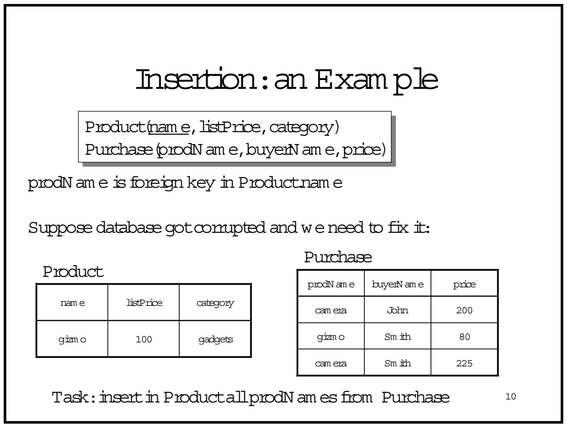
Missing attribute fi NULL.
May drop attribute names if give them in order.



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:

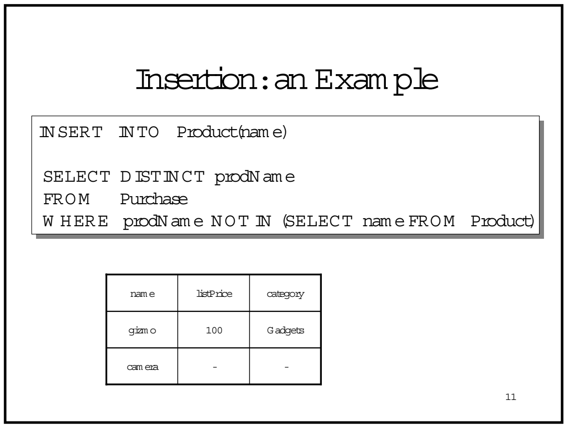
Product

name	listPrice	category
gizmo	100	gadgets

Purchase

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

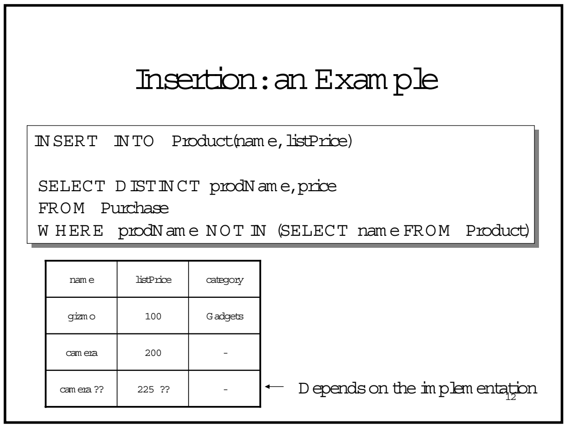
Task: insert in Product all prodNames from Purchase



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	-	-

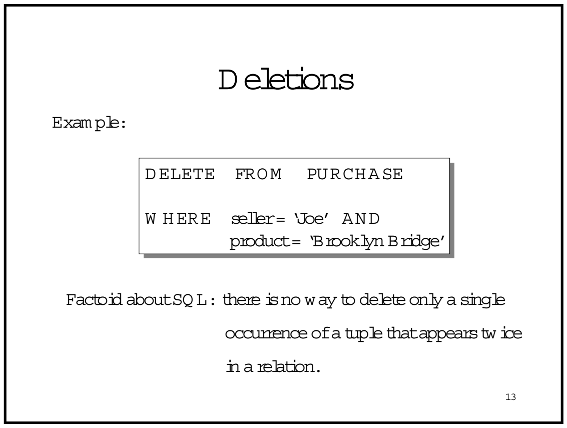


Insertion: an Example

```
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

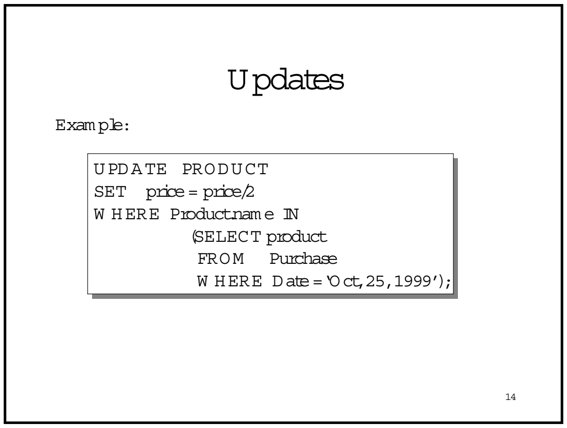


Deletions

Example:

```
DELETE  FROM  PURCHASE

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

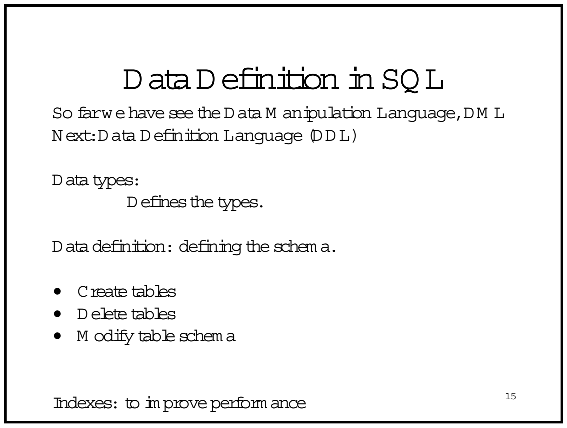
Factoid about SQL: there is no way to delete only a single occurrence of a tuple that appears twice in a relation.



Updates

Example:

```
UPDATE PRODUCT
SET  price = price/2
WHERE Productname IN
          (SELECT product
           FROM  Purchase
           WHERE Date = 'Oct, 25, 1999');
```



Data Definition in SQL

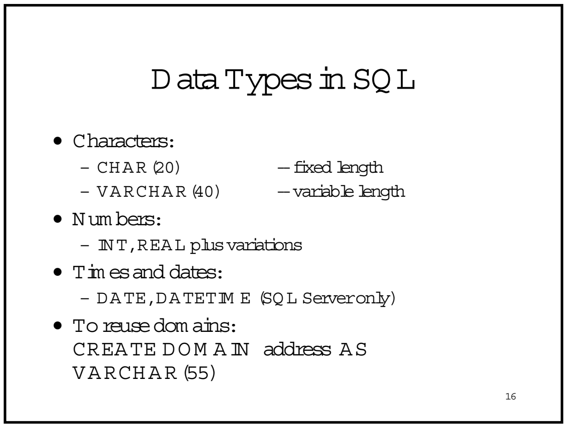
So far we have see the Data Manipulation Language, DML
Next: Data Definition Language (DDL)

Data types:
Defines the types.

Data definition: defining the schema.

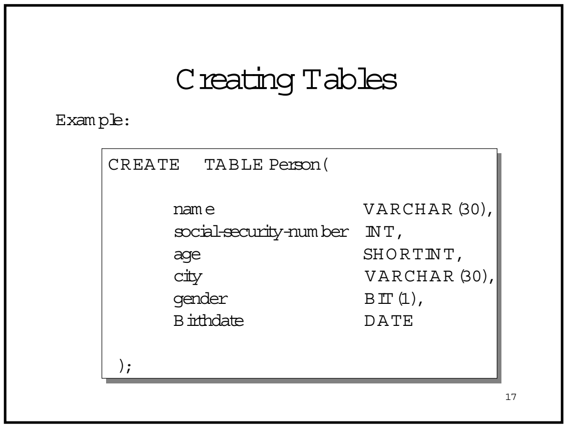
- Create tables
- Delete tables
- Modify table schema

Indexes: to improve performance



Data Types in SQL

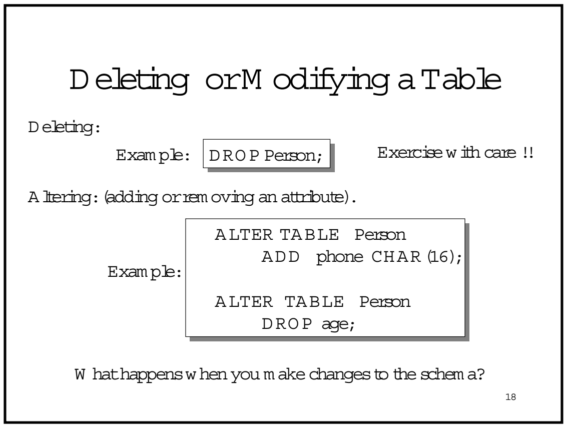
- Characters:
  - CHAR(20) – fixed length
  - VARCHAR(40) – variable length
- Numbers:
  - INT, REAL plus variations
- Times and dates:
  - DATE, DATETIME (SQL Server only)
- To reuse domains:
  CREATE DOMAIN address AS VARCHAR(55)



Creating Tables

Example:

```
CREATE  TABLE Person(

      name                    VARCHAR(30),
      social-security-number  INT,
      age                     SHORTINT,
      city                    VARCHAR(30),
      gender                  BIT(1),
      Birthdate               DATE

 );
```



Deleting or Modifying a Table

Deleting:

Example: `DROP Person;` Exercise with care !!

Altering: (adding or removing an attribute).

Example:

```
ALTER TABLE Person
    ADD  phone CHAR(16);

ALTER TABLE Person
    DROP  age;
```

What happens when you make changes to the schema?

Default Values

Specifying default values:

```
CREATE TABLE Person(
       name                  VARCHAR(30),
       social-security-number INT,
       age      SHORTINT  DEFAULT 100,
       city   VARCHAR(30) DEFAULT  'Seattle',
       gender      CHAR(1)  DEFAULT  '?',
       Birthdate              DATE
```

The default of defaults: NULL

Indexes

REALLY important to speed up query processing time.

Suppose we have a relation

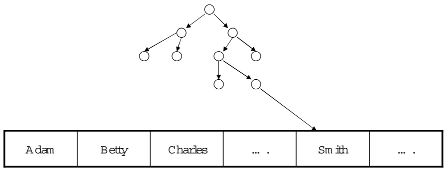
Person (name, age, city)

```
SELECT *
FROM   Person
WHERE  name = "Smith"
```

Sequential scan of the file Person may take long

Indexes

- Create an index on name:



Adam	Betty	Charles	….	Smith	….

- B+ trees have fan-out of 100s: max 4 levels !

Creating Indexes

Syntax:

```
CREATE INDEX  nameIndex ON Person(name)
```

Creating Indexes

Indexes can be created on more than one attribute:

Example:

```
CREATE INDEX doubleindex ON
               Person (age, city)
```

Helps in:

```
SELECT *
FROM   Person
WHERE age = 55 AND city = "Seattle"
```

But not in:

```
SELECT *
FROM   Person
WHERE city = "Seattle"
```

Creating Indexes

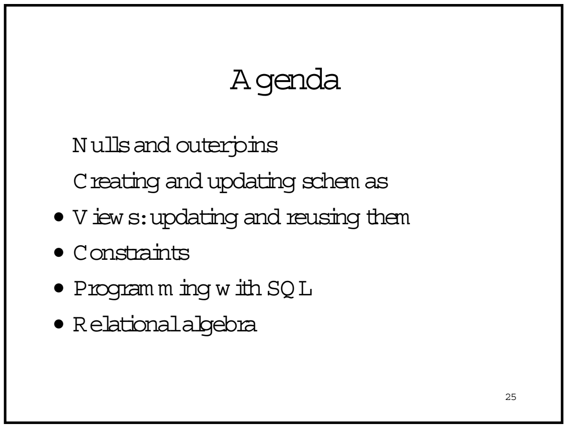
Indexes can be useful in range queries too:

```
CREATE INDEX ageIndex ON  Person (age)
```

B+ trees help in:

```
SELECT *
FROM Person
WHERE age > 25 AND age < 28
```

Why not create indexes on everything?



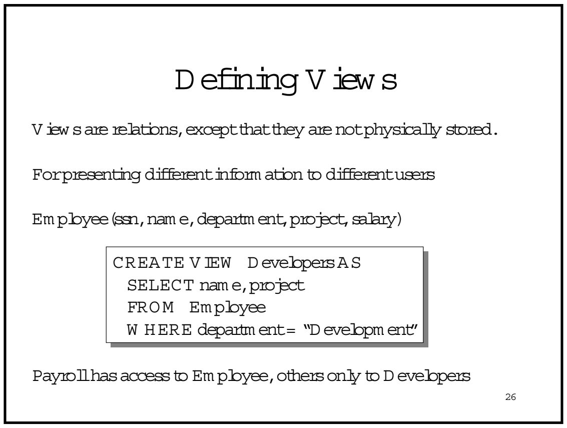
Agenda

Nulls and outerjoins

Creating and updating schemas

- Views: updating and reusing them
- Constraints
- Programming with SQL
- Relational algebra

25



Defining Views

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

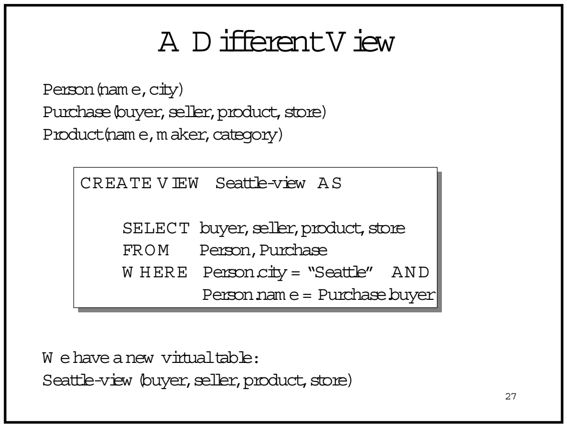
For presenting different information to different users

Employee(ssn, name, department, project, salary)

```
CREATE VIEW  Developers AS
  SELECT name, project
  FROM  Employee
  WHERE department = "Development"
```

Payroll has access to Employee, others only to Developers

26



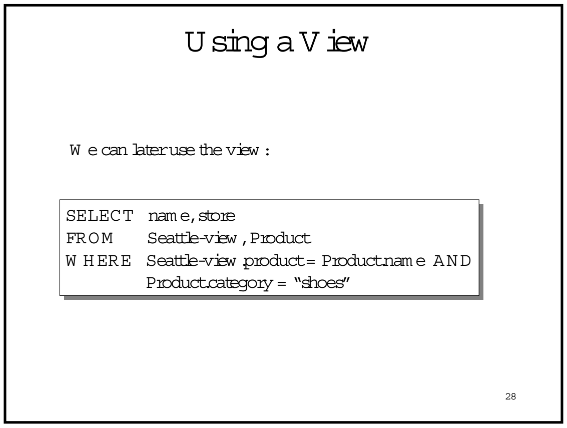
A Different View

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

```
CREATE VIEW  Seattle-view  AS

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

We have a new virtual table:
Seattle-view(buyer, seller, product, store)

27

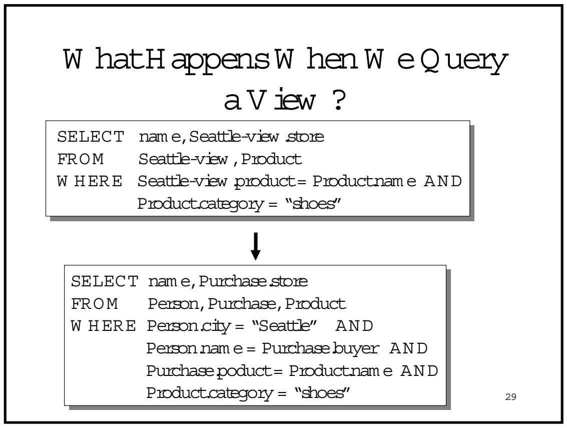


Using a View

We can later use the view:

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

28



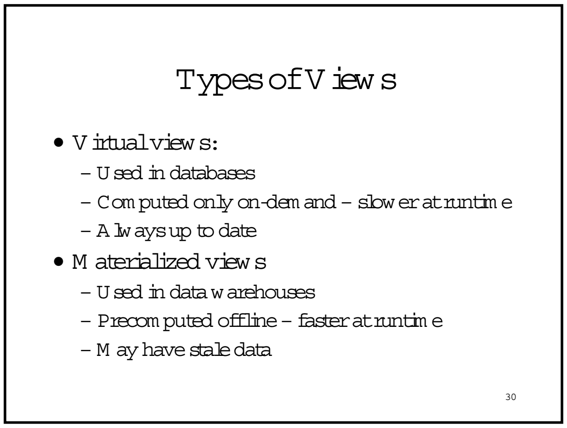
What Happens When We Query a View ?

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

↓

```
SELECT  name, Purchase.store
FROM    Person, Purchase, Product
WHERE   Person.city = "Seattle"  AND
        Person.name = Purchase.buyer AND
        Purchase.poduct = Product.name AND
        Product.category = "shoes"
```

29



Types of Views

- Virtual views:
  - Used in databases
  - Computed only on-demand – slower at runtime
  - Always up to date
- Materialized views
  - Used in data warehouses
  - Precomputed offline – faster at runtime
  - May have stale data

30

Updating Views

How can I insert a tuple into a table that doesn't exist?

Employee(ssn, name, department, project, salary)

```
CREATE VIEW Developers AS
  SELECT name, project
  FROM Employee
  WHERE department = “Development”
```

If we make the following insertion:

```
INSERT INTO Developers
VALUES(“Joe”, “Optimizer”)
```

It becomes:

```
INSERT INTO Employee
VALUES(NULL, “Joe”, “Development”, “Optimizer”, NULL)
```

Non-Updatable Views

```
CREATE VIEW Seattle-view AS

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

How can we add the following tuple to the view?

(“Joe”, “Shoe Model12345”, “Nine West”)

We need to add “Joe” to Person first, but we don't have all its attributes

Answering Queries Using Views

- What if we want to use a set of views to answer a query.
- Why?
  - The obvious reason…
  - Answering queries over web data sources.
- Very cool stuff! (i.e., I did a lot of research on this).

Reusing a Materialized View

- Suppose I have only the result of SeattleView:

```
SELECT buyer, seller, product, store
FROM   Person, Purchase
WHERE  Person.city = ‘Seattle’ AND
       Person.per-name = Purchase.buyer
```

- and I want to answer the query

```
SELECT buyer, seller
FROM   Person, Purchase
WHERE  Person.city = ‘Seattle’ AND
       Person.per-name = Purchase.buyer AND
       Purchase.product=‘gizmo’.
```

Then, I can rewrite the query using the view.

Query Rewriting Using Views

Rewritten query:

```
SELECT buyer, seller
FROM   SeattleView
WHERE  product= ‘gizmo’
```

Original query:

```
SELECT buyer, seller
FROM   Person, Purchase
WHERE  Person.city = ‘Seattle’ AND
       Person.per-name = Purchase.buyer AND
       Purchase.product=‘gizmo’.
```

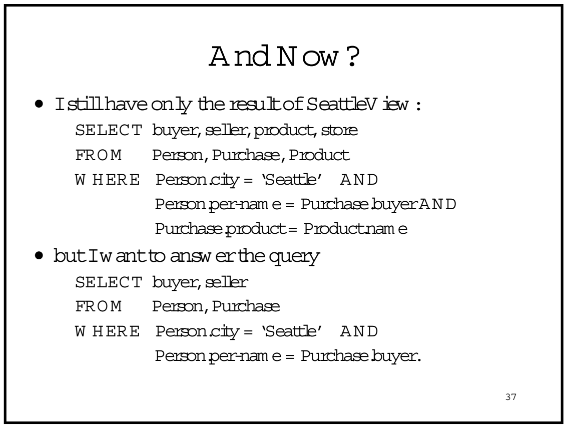
Another Example

- I still have only the result of SeattleView:

```
SELECT buyer, seller, product, store
FROM   Person, Purchase
WHERE  Person.city = ‘Seattle’ AND
       Person.per-name = Purchase.buyer
```

- but I want to answer the query

```
SELECT buyer, seller
FROM   Person, Purchase
WHERE  Person.city = ‘Seattle’ AND
       Person.per-name = Purchase.buyer AND
       Person.Phone LIKE ‘206 543 %’.
```



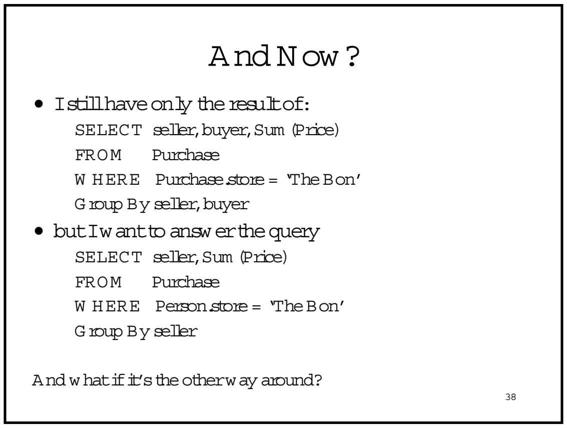
And Now?

- I still have only the result of SeattleView:

```
SELECT buyer, seller, product, store
FROM   Person, Purchase, Product
WHERE  Person.city = ‘Seattle’  AND
       Person.per-name = Purchase.buyer AND
       Purchase.product = Product.name
```

- but I want to answer the query

```
SELECT buyer, seller
FROM   Person, Purchase
WHERE  Person.city = ‘Seattle’  AND
       Person.per-name = Purchase.buyer.
```



And Now?

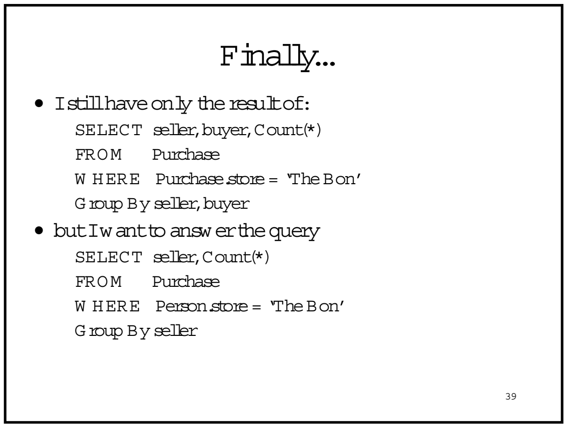
- I still have only the result of:

```
SELECT seller, buyer, Sum (Price)
FROM   Purchase
WHERE  Purchase.store = ‘The Bon’
Group By seller, buyer
```

- but I want to answer the query

```
SELECT seller, Sum (Price)
FROM   Purchase
WHERE  Person.store = ‘The Bon’
Group By seller
```

And what if it’s the other way around?



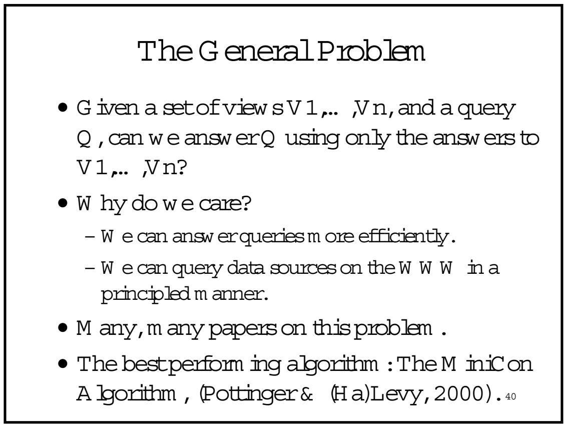
Finally…

- I still have only the result of:

```
SELECT seller, buyer, Count(*)
FROM   Purchase
WHERE  Purchase.store = ‘The Bon’
Group By seller, buyer
```

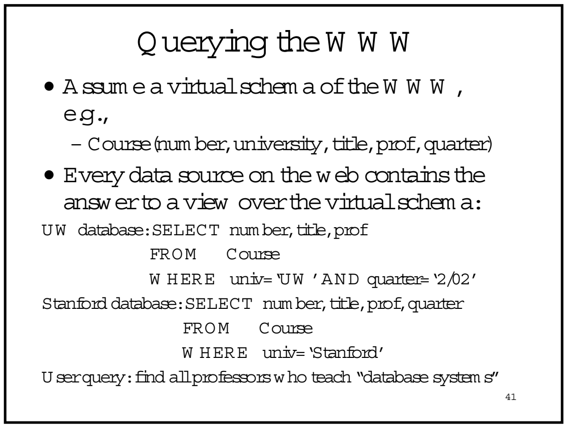
- but I want to answer the query

```
SELECT seller, Count(*)
FROM   Purchase
WHERE  Person.store = ‘The Bon’
Group By seller
```



The General Problem

- Given a set of views V1,… ,Vn, and a query Q, can we answer Q using only the answers to V1,… ,Vn?
- Why do we care?
  - We can answer queries more efficiently.
  - We can query data sources on the WWW in a principled manner.
- Many, many papers on this problem.
- The best performing algorithm: The MiniCon Algorithm, (Pottinger & (Ha)Levy, 2000).

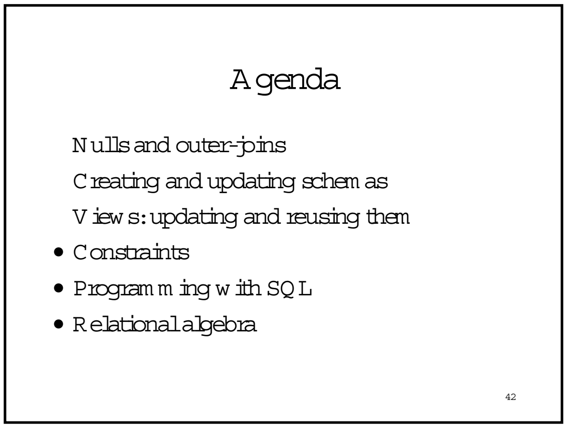


Querying the WWW

- Assume a virtual schema of the WWW, e.g.,
  - Course(number, university, title, prof, quarter)
- Every data source on the web contains the answer to a view over the virtual schema:

```
UW database: SELECT number, title, prof
             FROM   Course
             WHERE  univ=‘UW’ AND quarter=‘2/02’
Stanford database: SELECT number, title, prof, quarter
                   FROM   Course
                   WHERE  univ=‘Stanford’
```

User query: find all professors who teach “database systems”



Agenda

Nulls and outer-joins

Creating and updating schemas

Views: updating and reusing them

- Constraints
- Programming with SQL
- Relational algebra

Constraints in SQL

- A constraint = a property that we'd like our database to hold
- The system will enforce the constraint by taking some actions:
  - forbid an update
  - or perform compensating updates

Constraints in SQL

Constraints in SQL:

- Keys, foreign keys
- Attribute-level constraints
- Tuple-level constraints
- Global constraints: assertions

simplest

Most complex

The more complex the constraint, the harder it is to check and to enforce

Keys

```
CREATE TABLE Product (
      name CHAR(30) PRIMARY KEY,
      category VARCHAR(20))
```

OR:

```
CREATE TABLE Product (
      name CHAR(30),
      category VARCHAR(20)
PRIMARY KEY (name))
```

Keys with Multiple Attributes

```
CREATE TABLE Product (
      name CHAR(30),
      category VARCHAR(20),
      price INT,
   PRIMARY KEY (name, category))
```

Other Keys

```
CREATE TABLE Product (
      productID CHAR(10),
      name CHAR(30),
      category VARCHAR(20),
      price INT,
      PRIMARY KEY (productID),
      UNIQUE (name, category))
```

There is at most one PRIMARY KEY;
there can be many UNIQUE

Foreign Key Constraints

Referential integrity constraints

```
CREATE TABLE Purchase (
      prodName CHAR(30)
            REFERENCES Product(name),
      date DATETIME)
```

prodName is a foreign key to Product(name)
name must be a key in Product

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Foreign Key Constraints

- OR

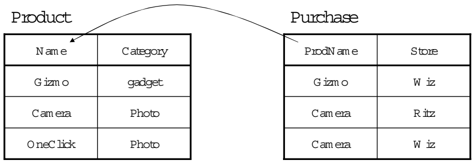
```
CREATE TABLE Purchase (
     prodName CHAR(30),
     category VARCHAR(20),
     date DATETIME,
     FOREIGN KEY (prodName, category)
       REFERENCES Product(name, category)
```

- (name, category) must be a PRIMARY KEY

What happens during updates?

Types of updates:

- In Purchase: insert/update
- In Product: delete/update



Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

What happens during updates?

- SQL has three policies for maintaining referential integrity:
- Reject violating modifications (default)
- Cascade: after a delete/update do a delete/update
- Set-null set foreign-key field to NULL

READING ASSIGNEMNT: 7.1.5, 7.1.6

Constraints on Attributes and Tuples

- Constraints on attributes:
  NOT NULL — obvious meaning...
  CHECK condition — any condition !
- Constraints on tuples
  CHECK condition

What is the difference from Foreign-Key ?

```
CREATE TABLE Purchase (
     prodName CHAR(30)
          CHECK (prodName IN
                   SELECT Product.name
                   FROM Product),
     date DATETIME NOT NULL)
```

General Assertions

```
CREATE ASSERTION myAssert CHECK
 NOT EXISTS(
    SELECT Product.name
    FROM Product, Purchase
    WHERE Product.name = Purchase.prodName
    GROUP BY Product.name
    HAVING count(*) > 200)
```

Final Comments on Constraints

- Can give them names, and alter later
  - Read in the book.
- We need to understand exactly when they are checked
- We need to understand exactly what actions are taken if they fail

Triggers

Enable the database programmer to specify:

- when to check a constraint,
- what exactly to do.

A trigger has 3 parts:

- An event (e.g., update to an attribute)
- A condition (e.g., a query to check)
- An action (deletion, update, insertion)

When the event happens, the system will check the constraint, and if satisfied, will perform the action.

NOTE: triggers may cause cascading effects.
Database vendors did not wait for standards with triggers!

Elements of Triggers (in SQL3)

- Timing of action execution: before, after or instead of triggering event
- The action can refer to both the old and new state of the database.
- Update events may specify a particular column or set of columns.
- A condition is specified with a WHEN clause.
- The action can be performed either for
  - once for every tuple, or
  - once for all the tuples that are changed by the database operation.

Example: Row Level Trigger

```
CREATE TRIGGER   NoLowerPrices

AFTER UPDATE OF  price  ON Product
REFERENCING
    OLD  AS OldTuple
    NEW  AS  NewTuple
WHEN (OldTuple.price > NewTuple.price)
    UPDATE  Product
    SET  price = OldTuple.price
    WHERE  name = NewTuple.name

FOR EACH ROW
```

Statement Level Trigger

```
CREATE TRIGGER  average-price-preserve
INSTEAD OF UPDATE OF price ON Product

REFERENCING
   OLD_TABLE  AS OldStuff
   NEW_TABLE AS NewStuff
WHEN (1000 <
          (SELECT  AVG (price)
           FROM ((Product EXCEPT OldStuff) UNION NewStuff))
DELETE  FROM Product
           WHERE (name, price, company) IN OldStuff;
INSERT INTO Product
  (SELECT  * FROM NewStuff)
```

Bad Things Can Happen

```
CREATE TRIGGER Bad-trigger

AFTER UPDATE OF price IN Product
REFERENCING OLD AS OldTuple
            NEW AS NewTuple

WHEN (NewTuple.price > 50)

   UPDATE Product
   SET price = NewTuple.price * 2
   WHERE name = NewTuple.name

FOR EACH ROW
```

Agenda

Nulls and outerjoins

Creating and updating schemas

Views: updating and reusing them

Constraints

- Programming with SQL
- Relational algebra

Embedded SQL

- direct SQL (= ad-hoc SQL) is rarely used
- in practice: SQL is embedded in some application code
- SQL code is identified by special syntax

Impedance Mismatch

- Example: SQL in C:
  - C uses int, char[..], pointers, etc
  - SQL uses tables
- Impedance mismatch = incompatible types

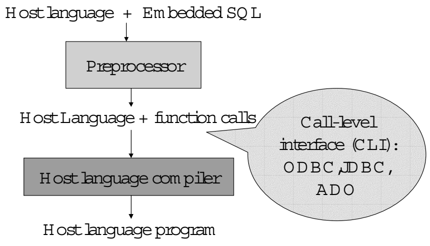
The Impedance Mismatch Problem

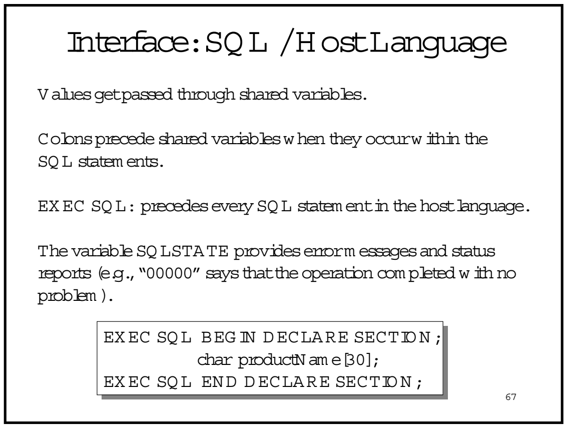
Why not use only one language?

- Forgetting SQL: "we can quickly dispense with this idea" [textbook, pg. 351].
- SQL cannot do everything that the host language can do.

Solution: use cursors

Programs with Embedded SQL


Host language + Embedded SQL
Preprocessor
Host Language + function calls
Host language compiler
Host language program
Call-level interface (CLI): ODBC, JDBC, ADO



Interface: SQL / Host Language

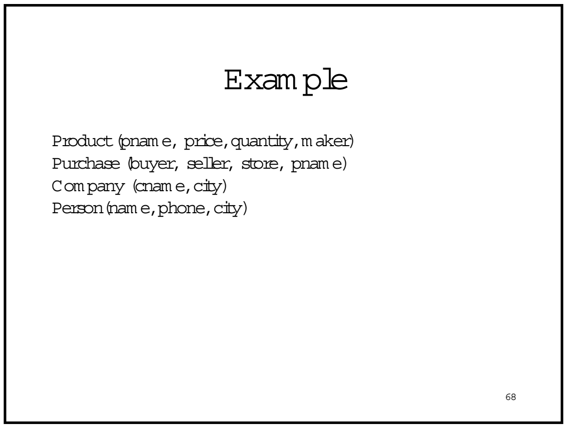
Values get passed through shared variables.

Colons precede shared variables when they occur within the SQL statements.

EXEC SQL: precedes every SQL statement in the host language.

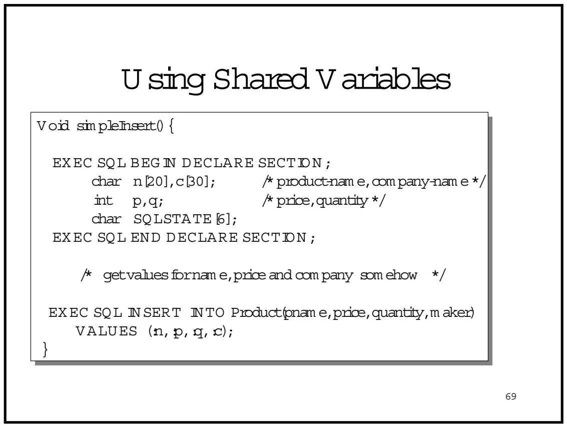
The variable SQLSTATE provides error messages and status reports (e.g., "00000" says that the operation completed with no problem).

```
EXEC SQL BEGIN DECLARE SECTION;
             char productName[30];
EXEC SQL END DECLARE SECTION;
```



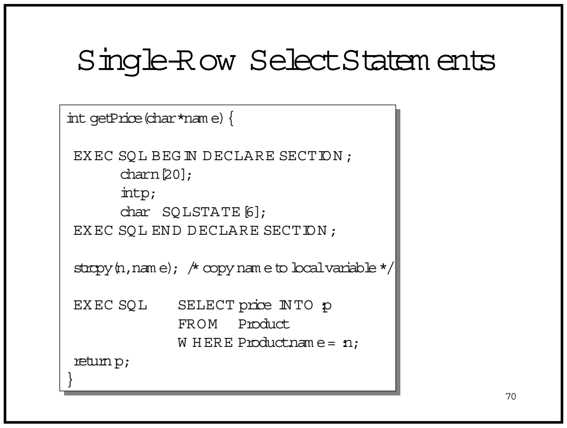
Example

Product (pname, price, quantity, maker)
Purchase (buyer, seller, store, pname)
Company (cname, city)
Person(name, phone, city)



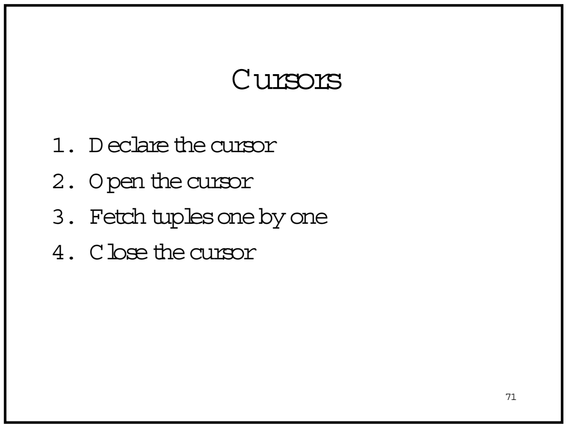
Using Shared Variables

```
Void simpleInsert() {

  EXEC SQL BEGIN DECLARE SECTION;
       char  n[20], c[30];       /* product-name, company-name */
       int   p, q;               /* price, quantity */
       char  SQLSTATE[6];
  EXEC SQL END DECLARE SECTION;

     /*  get values for name, price and company somehow   */

 EXEC SQL INSERT  INTO Product(pname, price, quantity, maker)
    VALUES (:n, :p, :q, :c);
}
```



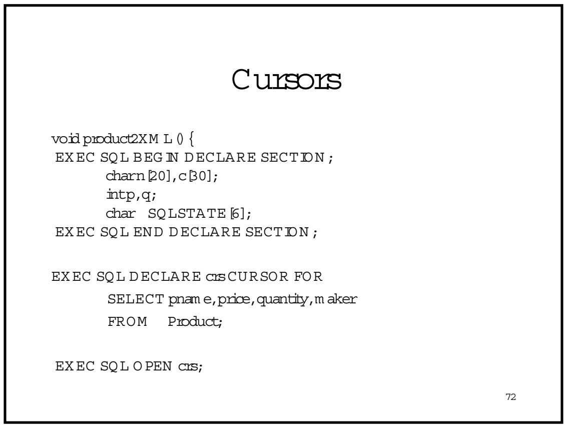
Single-Row Select Statements

```
int getPrice(char *name) {

 EXEC SQL BEGIN DECLARE SECTION;
       char n[20];
       int p;
       char  SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;

 strcpy(n, name);  /* copy name to local variable */

 EXEC SQL     SELECT price  INTO :p
              FROM    Product
              WHERE Product.name = :n;
 return p;
}
```



Cursors

1. Declare the cursor
2. Open the cursor
3. Fetch tuples one by one
4. Close the cursor



Cursors

```
void product2XML() {
 EXEC SQL BEGIN DECLARE SECTION;
        char n[20], c[30];
        int p, q;
        char  SQLSTATE[6];
 EXEC SQL END DECLARE SECTION;

EXEC SQL DECLARE crs CURSOR FOR
        SELECT pname, price, quantity, maker
        FROM   Product;

 EXEC SQL OPEN crs;
```

Cursors

```
    printf("<allProducts>\n");
    while (1) {
       EXEC SQL FETCH FROM crs INTO :n, :p, :q, :c;
       if (NO_MORE_TUPLES) break;
       printf("   <product>\n");
       printf("         <name>    %s   </name>\n",    n);
       printf("         <price>   %d  </price>\n",    p);
       printf("         <quantity> %d  </quantity>\n",q);
       printf("         <maker>   %s  </maker>\n",    c);
       printf("   </product>\n");
    }
    EXECT SQL CLOSE crs;
    printf("</allProducts>\n");
}
```

- What is NO_MORE_TUPLES ?

```
#define NO_MORE_TUPLES !(strcmp(SQLSTATE,"02000"))
```

More on Cursors

- cursors can modify a relation as well as read it.
- We can determine the order in which the cursor will get tuples by the ORDER BY keyword in the SQL query.
- Cursors can be protected against changes to the underlying relations.
- The cursor can be a scrolling one: can go forward, backward +n, -n, Abs(n), Abs(-n).

Agenda

Nulls and outerjoins

Creating and updating schemas

Views: updating and reusing them

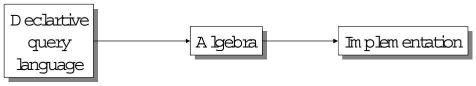
Constraints

Programming with SQL

- Relational algebra

Relational Algebra

- Formalism for creating new relations from existing ones
- Its place in the big picture:



SQL, relational calculus

Relational algebra
Relational bag algebra

Relational Algebra

- Five operators:
  - Union: ˝
  - Difference: -
  - Selection: s
  - Projection: P
  - Cartesian Product: ·
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: r

1. Union and 2. Difference

- R1 ¨ R2
- Example:
  - ActiveEmployees ¨ RetiredEmployees
- R1 – R2
- Example:
  - AllEmployees – RetiredEmployees

What about Intersection ?

- It is a derived operator
- R1 ˙ R2 = R1 – (R1 – R2)
- Also expressed as a join (will see later)
- Example
  - UnionizedEmployees ˙ RetiredEmployees

3. Selection

- Returns all tuples which satisfy a condition
- Notation: $s_c(R)$
- Examples
  - $s_{Salary > 40000}$ (Employee)
  - $s_{name = "Smithh"}$ (Employee)
- The condition c can be =, <, £, >, ‡, <>

Selection Example

Employee

SSN	Name	DepartmentID	Salary
999999999	John	1	30,000
777777777	Tony	1	32,000
888888888	Alice	2	45,000

Find all employees with salary more than $40,000.

$s_{Salary > 40000}$ (Employee)

SSN	Name	DepartmentID	Salary
888888888	Alice	2	45,000

4. Projection

- Eliminates columns, then removes duplicates
- Notation: $P_{A1,\ldots An}$ (R)
- Example: project social-security number and names:
  - $P_{SSN, Name}$ (Employee)
  - Output schema: Answer(SSN, Name)

Projection Example

Employee

SSN	Name	DepartmentID	Salary
999999999	John	1	30,000
777777777	Tony	1	32,000
888888888	Alice	2	45,000

$\mathbf{P}_{SSN, Name}$ (Employee)

SSN	Name
999999999	John
777777777	Tony
888888888	Alice

5. Cartesian Product

- Each tuple in R1 with each tuple in R2
- Notation: R1 · R2
- Example:
  - Employee · Dependents
- Very rare in practice; mainly used to express joins

Cartesian Product Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependents

EmployeeSSN	Dname
999999999	Emily
777777777	Joe

Employee x Dependents

Name	SSN	EmployeeSSN	Dname
John	999999999	999999999	Emily
John	999999999	777777777	Joe
Tony	777777777	999999999	Emily
Tony	777777777	777777777	Joe

Relational Algebra

- Five operators:
  - Union: ¨
  - Difference: -
  - Selection: s
  - Projection: P
  - Cartesian Product: ·
- Derived or auxiliary operators:
  - Intersection, complement
  - Joins (natural,equi-join, theta join, semi-join)
  - Renaming: r

Renaming

- Changes the schema, not the instance
- Notation: $r_{B1,\ldots,Bn}$ (R)
- Example:
  - $r_{LastName, SocSocNo}$ (Employee)
  - Output schema: Answer(LastName, SocSocNo)

Renaming Example

Employee

Name	SSN
John	999999999
Tony	777777777

$r_{LastName,SocSocNo}$ (Employee)

LastName	SocSocNo
John	999999999
Tony	777777777

Natural Join

- Notation: R1 ⋈ R2
- Meaning: R1 ⋈ R2 = P_A (s_C (R1 · R2))
- Where:
  - The selection s_C checks equality of all common attributes
  - The projection eliminates the duplicate common attributes

Natural Join Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependents

SSN	Dname
999999999	Emily
777777777	Joe

Employee $\bowtie$ Dependents =
$P_{Name,SSN,Dname}(s_{SSN=SSN2}(Employee \; x \; r_{SSN2,Dname}(Dependents)))$

Name	SSN	Dname
John	999999999	Emily
Tony	777777777	Joe

Natural Join

- R =

A	B
X	Y
X	Z
Y	Z
Z	V

S =

B	C
Z	U
V	W
Z	V

- R $\bowtie$ S=

A	B	C
X	Z	U
X	Z	V
Y	Z	U
Y	Z	V
Z	V	W

Natural Join

- Given the schemas R(A, B, C, D), S(A, C, E), what is the schema of R $\bowtie$ S ?
- Given R(A, B, C), S(D, E), what is R $\bowtie$ S ?
- Given R(A, B), S(A, B), what is R $\bowtie$ S ?

Theta Join

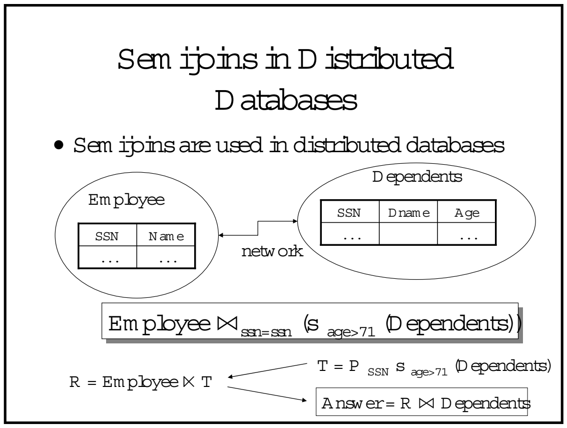
- A join that involves a predicate
- $R1 \bowtie_q R2 = s_q (R1 \cdot R2)$
- Here q can be any condition

Eq-join

- A theta join where q is an equality
- $R1 \bowtie_{A=B} R2 = s_{A=B} (R1 \cdot R2)$
- Example:
  - Employee $\bowtie_{SSN=SSN}$ Dependents
- Most useful join in practice

Semijoin

- $R \ltimes S = P_{A1,\ldots,An} (R \bowtie S)$
- Where $A_1, \ldots, A_n$ are the attributes in R
- Example:
  - Employee $\ltimes$ Dependents



Semijoins in Distributed Databases

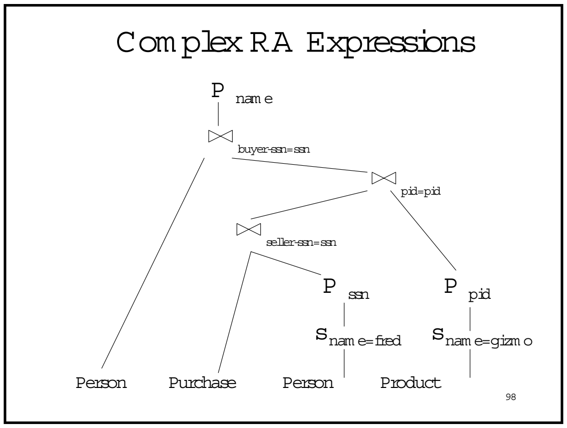
- Semijoins are used in distributed databases

Employee $\bowtie_{ssn=ssn}$ (s $_{age>71}$ (Dependents))

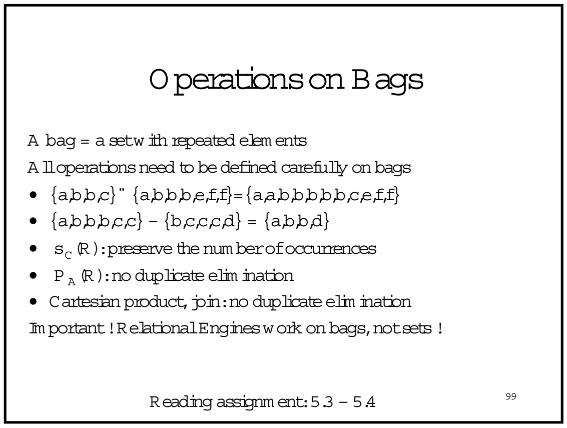
T = P $_{SSN}$ s $_{age>71}$ (Dependents)

R = Employee ⋉ T

Answer = R ⋈ Dependents



Complex RA Expressions



Operations on Bags

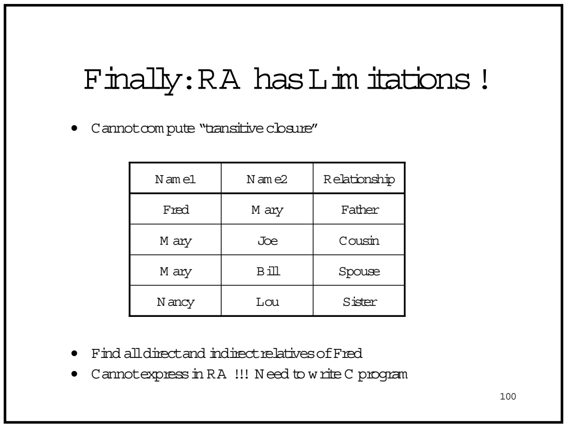
A bag = a set with repeated elements

All operations need to be defined carefully on bags

- {a,b,b,c}" {a,b,b,b,e,f,f}={a,a,b,b,b,b,b,c,e,f,f}
- {a,b,b,b,c,c} – {b,c,c,c,d} = {a,b,b,d}
- s $_C$ (R): preserve the number of occurrences
- P $_A$ (R): no duplicate elimination
- Cartesian product, join: no duplicate elimination

Important ! Relational Engines work on bags, not sets !

Reading assignment: 5.3 – 5.4



Finally: RA has Limitations !

- Cannot compute "transitive closure"

Name1	Name2	Relationship
Fred	Mary	Father
Mary	Joe	Cousin
Mary	Bill	Spouse
Nancy	Lou	Sister

- Find all direct and indirect relatives of Fred
- Cannot express in RA !!! Need to write C program