CSE 215:SQL

Dr. Mohammed Eunus Ali
(eunus@cse.buet.ac.bd)

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000, Bangladesh

(with some slides integrated from those of Jenifer Widom, Alon Halevy, Carlo Curino, and Michael Stonebraker.)

SQL Introduction

- Standard language for querying and manipulating data

  Structured Query Language (SQL)

- **Many standards out there: SQL92, SQL2, SQL3, SQL99**

- **Vendors support various subsets of these**

SQL

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

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

Selections

Company(sticker, name, country, stockPrice)

Find all US companies whose stock is > 50:

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

Output schema: R(sticker, name, country, stockPrice)

Selections

- **What you can use in WHERE:**

  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.

Schema

- **Product (pname, price, category, maker)**
- **Purchase (buyer, seller, store, product)**
- **Company (cname, stockPrice, country)**
- **Person(pname, phoneNumber, city)**

Projections

- **Select only a subset of the attributes**

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

- **Input schema: Company(sticker, name, country, stockPrice)**
- **Output schema: R(name, stock price)**

Projections

Rename the attributes in the resulting table

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

Input schema: Company(sticker, name, country, stockPrice)
Output schema: R(company, price)

Joins

Find names of people living in Seattle that bought gizmo products, and the names of the stores they bought from

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

Disambiguating Attributes

Find names of people buying telephony products:

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

```
SELECT   Person.name
FROM     Person, Purchase, Product
WHERE    Person.name=Purchase.buyer
    AND  Person.name=Product.name
    AND  Product.category=“telephony”
```

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)

Tuple Variables

Tuple variables introduced automatically by the system:

Product (name, price, category, maker)

```
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 explicitly.

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 U {(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 U {(a1,…,ak)}
return Answer

Doesn't impose any order !
Like Datalog

Meaning (Semantics) of SQL Queries

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

3. Translation to Datalog: one rule

$$\text{Answer}(a_1,\ldots,a_k) \leftarrow R_1(x_{11},\ldots,x_{1p}),\ldots,R_n(x_{n1},\ldots,x_{np}), \text{Conditions}$$

Meaning (Semantics) of SQL Queries

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

4. Translation to Relational algebra:

$$\Pi_{a1,\ldots,ak} (\sigma_{Conditions} (R1 \times R2 \times \ldots \times Rn))$$

Select-From-Where queries are precisely Select-Project-Join

Movie Database Schema

Schema

Movie (title, year, length, inColor, studioName, producerC#)
StarsIn (movieTitle ,movieYear, starName)
MovieStar(name, address, gender, birthdate)
MovieExec(name, address, cert#, netWorth)
Studio(name, address, presC#)

Ordering the Output

Show all Disney movies of 1990 in ascending order of their lengths.

Movie (title, year, length, inColor, studioName, producerC#)

```
SELECT   name, address, length
FROM     Movie
WHERE    studioNmae = 'Disney' AND year=1990
ORDER BY     length;
```

If lengths are equal then in alphabetical order.

Union, Intersection, Difference

Find the name and address of all female movie stars who are Also movie executives with a net wroth over $100,000,000.

```
(SELECT  name, address
 FROM    MovieStar
 WHERE   gender="F")

    INTERSECTION

(SELECT  name, address
 FROM    MovieExec
 WHERE   networth > 100,000,000
```

You must have the same attribute names (otherwise: rename).

Union, Intersection, Difference

Find the names and addresses of movie stars who are movie executives.

```
(SELECT  name, address
 FROM    MovieStar)

EXCEPT

(SELECT  name, address
 FROM    MovieExec)
```

Union, Intersection, Difference

Find the titles and years of movies that appered in either the Movie or StarsIn

```
(SELECT  title, year
 FROM    Movie)

EXCEPT

(SELECT  movieTtile As title, movieYear as year
 FROM    StarsIn)
```

You must have the same attribute names (otherwise: rename).

Subqueries

- **A query that is a part of another query is called subquery.**
- **Subqueries can be used in WHERE or FROM clauses**

E.g., Show the producer name of Star Wars:

```
SELECT name
FROM    MovieExec
WHERE  cert# =
         (SELECT  producerC#
           FROM     Movie
           WHERE   title = 'Star Wars');
```

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 name
FROM   Movie, MovieExec
WHERE  title = 'Star Wars' AND producerC# = cert#
```

Is this query equivalent to the previous one ?

Subqueries Returning Relations

Find studioname of Harrison Ford Movie.

```
SELECT  studioname
FROM    Movie
WHERE   (name,year) IN
            (SELECT movieTitle, movieYear
             FROM   StartsIn
             WHERE  starName = “Harrison Ford”);
```

Here the subquery returns a set of values.

Find the producer names of Harrison Fords’ movies.

Subqueries Returning Relations

You can also use:

s IN R

s NOT IN R

s > ALL R

s > ANY R

EXISTS R

Correlated Queries

Movie (<u>title, year</u>, director, length)

Find movies whose title appears more than once.

correlation

```
SELECT  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

Subqueries in FORM clause

```
SELECT  studioname
FROM    Movie
WHERE   (name,year) IN
            (SELECT movieTitle, movieYear
             FROM   StartsIn
             WHERE  starName = “Harrison Ford”);
```

Removing Duplicates

Find the producer names of Harrison Fords' movies (one producer name should appear only once.)

```
SELECT  name
 FROM     MovieExec
 WHERE   cert# IN
      SELECT  producerC#
      FROM     Movie
      WHERE   (name,year) IN
                 (SELECT movieTitle, movieYear
                  FROM   StartsIn
                  WHERE  starName = "Harrison Ford");
```

Aggregation

Schema:

Product (pname, price, category, maker)
Purchase (buyer, seller, store, product)
Company (cname, stockPrice, country)
Person(pname, phoneNumber, city)

```
SELECT  Sum(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(name, category)        same as Count(*)
FROM    Product
WHERE   year > 1995
```

Better:

```
SELECT  Count(DISTINCT name, 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

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
```

Grouping and Aggregation

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.

First compute the relation (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
```

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 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

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

Evaluation steps:

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

Aggregation

Author(<u>login</u>,name)

Document(<u>url</u>, title)

Wrote(login,url)

Mentions(url,word)

- **Find all authors who wrote at least 10 documents:**

```
Select author.name
From author, wrote
Where author.login=wrote.login
Groupby author.name
Having count(wrote.url) > 10
```

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

```
Select author.name
From author, wrote, mentions
Where author.login=wrote.login and wrote.url=mentions.url
Groupby author.name
Having count(distinct mentions.word) > 10000
```

Exercises

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

Ex #1: Find people who bought telephony products.
Ex #2: Find names of people who bought American products
Ex #3: Find names of people who bought American products and did not buy French products
Ex #4: How much money did Fred spend on purchases?
Ex #5: What is the number and sum of the product sales by country of origin?

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 $\rightarrow$ 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');
```

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

- Character strings (fixed of varying length)
- Bit strings (fixed or varying length)
- Integer (SHORTINT)
- Floating point
- Dates and times

Domains (=types) will be used in table declarations.

To reuse domains:

```
CREATE DOMAIN  address  AS  VARCHAR(55)
```

Creating Tables

Example:

```
CREATE TABLE Person(

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

);
```

Deleting or Modifying a Table

Deleting:

Example:
```
DROP Person;
```

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  INTEGER,
        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

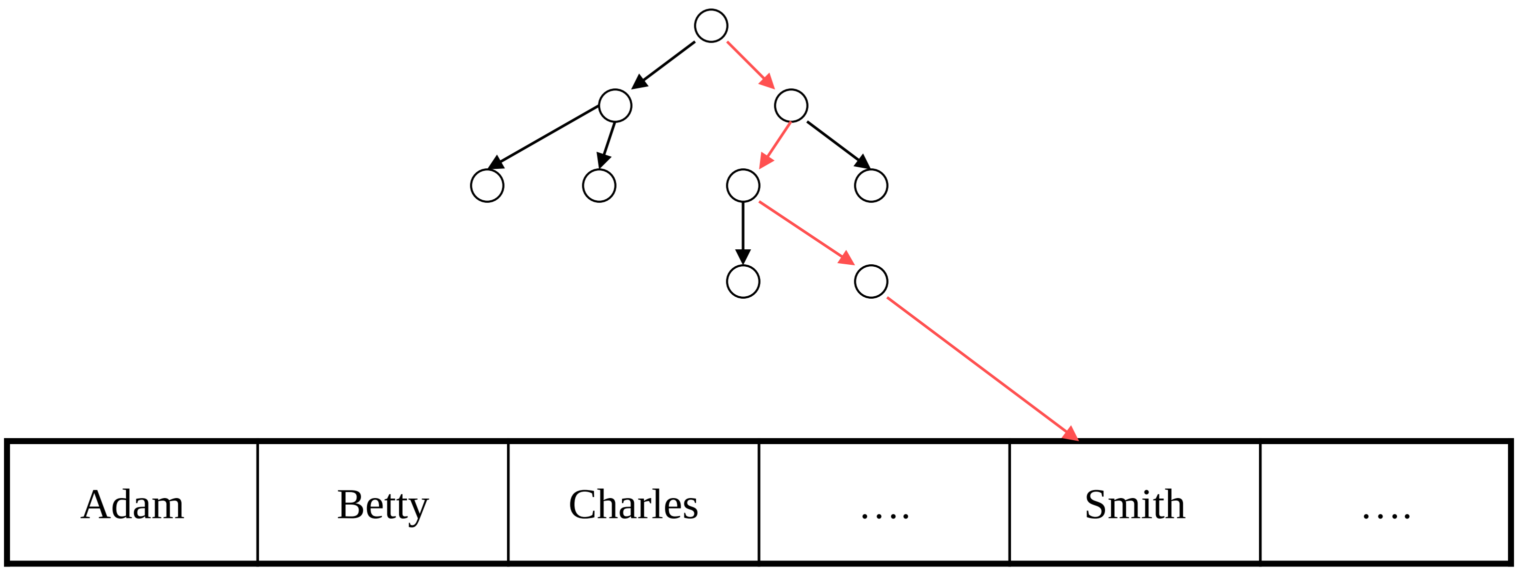
Person (name, age, city)

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

Sequential scan of the file Person may take long

Indexes

- **Create an index on name:**



- **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?

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

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)

A Different View

We can later use the view:

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

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.product = Product.name  AND
        Product.category = “shoes”
```

Types of Views

- **Virtual views:**
  - Used in databases
  - Computed only on-demand – slow at runtime
  - Always up to date
- **Materialized views**
  - Used in data warehouses (but recently also in DBMS)
  - Precomputed offline – fast at runtime
  - May have stale data

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”, NULL, “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 Model 12345", "Nine West")

What do we put in the Person.name and Purchase.buyer colum

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
WHERE   Person.city = 'Seattle'   AND
        Person.per-name = Purchase.buyer
```

- but I want to answer the query

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

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"

Null Values and Outerjoins

- **If x=Null then 4*(3-x)/7 is still NULL**

- **If x=Null then x="Joe" is UNKNOWN**
- **Three boolean values:**
  - FALSE = 0
  - UNKNOWN = 0.5
  - TRUE = 1

M.W.: NULL is not equal to NULL in SQL.

Null Values and Outerjoins

- 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)
```

Rule in SQL: include only tuples that yield TRUE

Null Values and Outerjoins

Unexpected behavior:

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

Some Persons are not included !

Null Values and Outerjoins

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

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	-

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

SQL: Constraints and Triggers

- Chapter 6 Ullman and Widom
- Certain properties we'd like our database to hold
- Modification of the database may break these properties
- Build handlers into the database definition
- Key constraints
- Referential integrity constraints.

Declaring a Primary Keys in SQL

```
CREATE TABLE MovieStar (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1));
```

OR:

```
CREATE TABLE MovieStar (
    name CHAR(30),
    address VARCHAR(255),
    gender CHAR(1)
    PRIMARY KEY (name));
```

Primary Keys with Multiple Attributes

```
CREATE TABLE MovieStar (
    name CHAR(30),
    address VARCHAR(255),
    gender CHAR(1),
    PRIMARY KEY (name, address));
```

Other Keys

```
CREATE TABLE MovieStar (
   name CHAR(30),
   address VARCHAR(255),
    phone  CHAR(10)  UNIQUE,
    gender CHAR(1),
    petName  CHAR(50),
    PRIMARY KEY (name),
    UNIQUE (gender, petName));
```

Foreign Key Constraints

```
CREATE TABLE ActedIn (
    Name CHAR(30) PRIMARY KEY,
    MovieName CHAR(30)
        REFERENCES Movies(MovieName),
    Year INT);
```

Foreign key constraint doesn't prevent null, as it's necessary in self relation (If trigger to ensure insert, make FK null & warning).

Foreign Key Constraints

- OR

```
CREATE TABLE ActedIn (
    Name CHAR(30) PRIMARY KEY,
    MovieName CHAR(30),
    Year INT,
    FOREIGN KEY MovieName
            REFERENCES Movies(MovieName)
```

- MovieName must be a PRIMARY KEY

How do we Maintain them?

- **Given a change to DB, there are several possible violations:**
  - Insert new tuple with bogus foreign key value
  - Update a tuple to a bogus foreign key value
  - Delete a tuple in the referenced table with the referenced foreign key value
  - Update a tuple in the referenced table that changes the referenced foreign key value

How to Maintain?

- **Recall, ActedIn has FK MovieName...**

  Movies(<u>MovieName</u>, year)

  (Fatal Attraction, 1987)

  ActedIn(ActorName, MovieName)

  (Michael Douglas, Fatal Attraction)

  insert: (Rick Moranis, Strange Brew)

How to Maintain?

- **Policies for handling the change…**
  - Reject the update (default)
  - Cascade (example: cascading deletes)
  - Set NULL
- **Can set update and delete actions independently in CREATE TABLE**

MovieName CHAR(30)

REFERENCES Movies(MovieName))

ON DELETE SET NULL

ON UPDATE CASCADE