Principles of Database Systems CSE 544p

Lecture #1

January 6th, 2011

Staff

- Instructor: Dan Suciu
  - CSE 662, suciu@cs.washington.edu
    Office hours: Tuesdays, 1:30-2:30

- TAs:
  - Prasang Upadhyaya,
    prasang@cs.washington.edu

Class Format

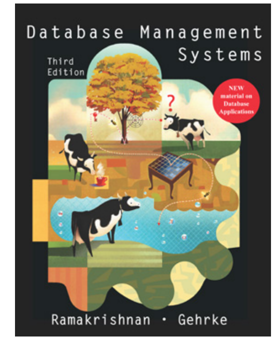
- Lectures Tuesday/Thursday 12-1:20
- Reading assignments
- 3 Homework Assignments
- A mini-research project

Announcements

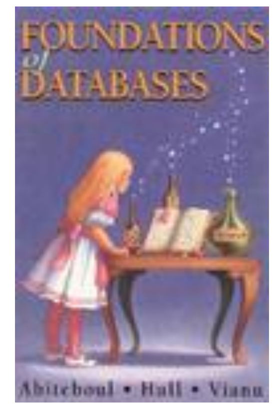
- No classes on:
  - January 4 (Tuesday)
  - January 18 (Tuesday)
  - March 10 (Thursday)

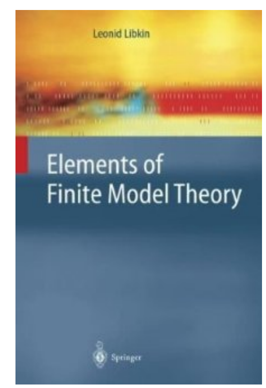
- We will make up for these classes; watch for further announcements

Textbook and Papers



- Official Textbook:
  - Database Management Systems. **3rd Ed.**, by Ramakrishnan and Gehrke. McGraw-Hill.
  - Book available on the Kindle too
  - Use it to read background material
- Other Books
  - Foundations of Databases, by Abiteboul, Hull, Vianu
  - Finite Model Theory, by Libkin





Textbook and Papers

- About 8 papers to read
  - Mix of old seminal papers and new papers
  - Papers available online on class website
  - Most papers available on Kindle
  - Some papers come from the "red book" [no need to get it]

Resources

- Web page: http://www.cs.washington.edu/education/courses/cse544/11wi/
  - Lectures will be available here
  - Reading assignments, papers
  - Homework assignments
  - Announcements about the projects
- Mailing list (see Webpage):
  - Announcements, group discussions
  - Please subscribe

Content of the Class

- Foundations
  – SQL, Relational calculus, Data Models, Views, Transactions
- Systems
  – Storage, query execution, query optimization, size estimation, parallel data processing
- Advanced Topics
  – Query languages and complexity classes, query containment, semijoin reductions, datalog (fixpoint semantics, magic sets, negation, modern applications of datalog), data provenance, data privacy, probabilistic databases

Goals of the Class

This is a CSE graduate level class !

- Goal:
  - Familiarity with database systems (postgres)
  - Appreciation of the impact of theory
  - Comfort in using data management in your research
- Goal:
  - Study key algorithms/techniques for massive data processing/analysis (sequential and/or parallel)
- Goal:
  - Exposure to some modern data management topics (provenance, privacy, probabilistic data)

Evaluation

- Class participation 10%
  - Paper readings and discussions
- Paper reviews 15%: Individual
  - Due by the beginning of each lecture
  - Reading questions are posted on class website
- Assignments 45%:
  - HW1: Using a DBMS (SQL, views, indexes, etc.)
  - HW2: Building a simple DBMS (groups of 1-2)
  - HW3: Theory
- Project 30%: Groups of 1-3
  - Small research or engineering. Start thinking now!

Class Participation 10%

- An important part of your grade
- Because
  - We would like you to study the material, read papers, and think about the topics throughout the quarter
- Expectations
  - Ask questions, raise issues, think critically
  - Learn to express your opinion
  - Respect other people's opinions

Paper Reviews 15%

- Between 1/2 page and 1 page in length
  - Summary of the main points of the paper
  - Critical discussion of the paper
- Reading questions
  - For some papers, we will post reading questions to help you figure out what to focus on when reading the paper
  - Please address these questions in your reviews
- Grading: credit/no-credit
  - You can skip one review without penalty
  - MUST submit review BEFORE lecture
  - Individual assignments (but feel free to discuss paper with others)

Assignments 45%

- HW1: Posted already on the Website
  - Install postres on your computer
  - Download a fun research data set (NELL)
  - Setup your NELL database
  - Practice SQL, relational calculus, views, constraints
- HW2: Build a simple DBMS
- HW3: Theory

We will accept late assignments with valid excuse

Project 30%

- Teams: 1-3 students
- Topics: choose one of:
  - A list of mini-research topics (see Website)
  - Come up with your own (related to your own research or interests, but must be related to databases; must involve either research or significant engineering)
- Deliverables (see Website for dates)
  - Project proposal
  - Milestone report
  - Final presentation
  - Final report
- Amount of work may vary widely between groups

Agenda for Today

- Brief overview of a traditional database systems
- SQL

For Tuesday: please read the slides on SQL;

Skip the parts on the Relational Calculus and Monotone Queries – we will discuss them on Tuesday

Databases

What is a database ?

Give examples of databases

Databases

What is a database ?

- A collection of files storing related data

Give examples of databases

- Accounts database; payroll database; UW's students database; Amazon's products database; airline reservation database

Database Management System

What is a DBMS ?

Give examples of DBMS

Database Management System

What is a DBMS ?

- A big C program written by someone else that allows us to manage efficiently a large database and allows it to persist over long periods of time

Give examples of DBMS

- DB2 (IBM), SQL Server (MS), Oracle, Sybase
- MySQL, Postgres, …

Market Shares

From 2006 Gartner report:

- IBM: 21% market with $3.2BN in sales
- Oracle: 47% market with $7.1BN in sales
- Microsoft: 17% market with $2.6BN in sales

An Example

The Internet Movie Database
http://www.imdb.com

- Entities:
  Actors (800k), Movies (400k), Directors, …
- Relationships:
  who played where, who directed what, …

Note

- In other classes (444, 544p):
  - We use IMDB/SQL Server for extensive practice of SQL
- In 544:
  - We will use NELL/postgres, which is more hands-on and more researchy
- If you want to practice more SQL:
  - Let me know and I will arrange for you to have access to the IMDB database and/or to SQL Server.

Tables

Actor:

id	fName	lName	gender
195428	Tom	Hanks	M
645947	Amy	Hanks	F
. . .			

Cast:

pid	mid
195428	337166
. . .	

Movie:

id	Name	year
337166	Toy Story	1995
. . .	. . .	. ..

SQL

```
SELECT *
FROM  Actor
```

SQL

```
SELECT count(*)
FROM  Actor
```

This is an *aggregate query*

SQL

```
SELECT *
FROM  Actor
WHERE lName = 'Hanks'
```

This is a *selection query*

SQL

```
SELECT *
FROM  Actor, Casts, Movie
WHERE lname='Hanks' and Actor.id = Casts.pid
  and Casts.mid=Movie.id and Movie.year=1995
```

This query has *selections* and *joins*

817k actors, 3.5M casts, 380k movies;

How can it be so fast ?

How Can We Evaluate the Query ?

Actor:

id	fName	lName	gender
. . .		Hanks	
. . .			

Cast:

pid	mid
. . .	
. . .	

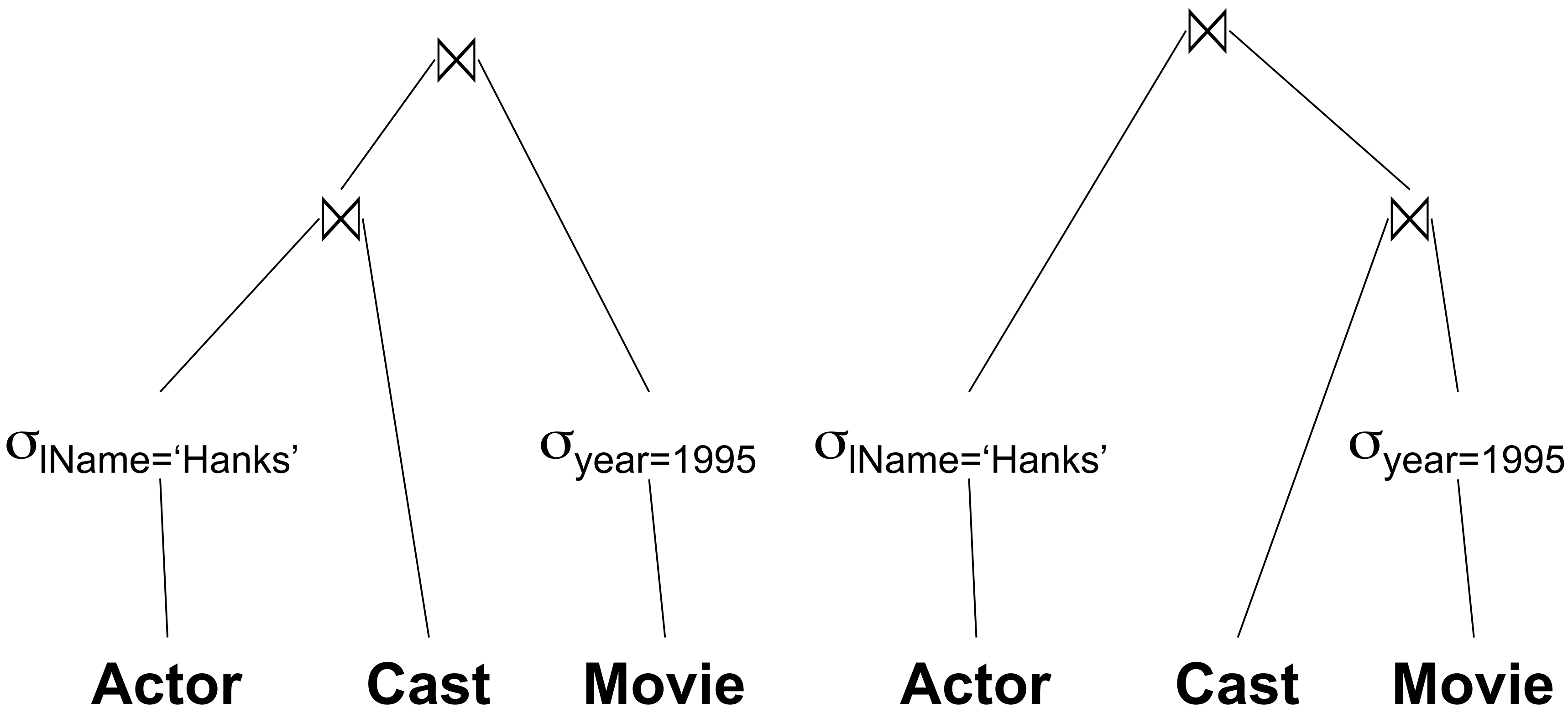
Movie:

id	Name	year
. . .		1995
. . .		

Plan 1: . . . . [in class]

Plan 2: . . . . [in class]

Evaluating Tom Hanks



Optimization and Query Execution

- Indexes: on Actor.lName, on Movie.year
- Query optimization
  - Access path selection
  - Join order
- Statistics
- Multiple implementations of joins

Types of Usages for Databases

- OLTP (online-transaction-processing)
  - Many updates: transactions are critical
  - Many “point queries”: retrieve the record with a given key.
- Decision-Support
  - Many aggregate/group-by queries.
  - Sometimes called *data warehouse*

Take-home Message 1

- Translating WHAT to HOW:
  - SQL = query language = WHAT we want
  - Relational algebra = algorithm = HOW to get it
  - In essence, RDBMS are about translating WHAT to HOW

- Query languages capture complexity classes:
  - Query languages = WHAT; complexity class =HOW
  - Examples:
    - Relational calculus = AC^0
    - Relational calculus + transitive closure = LOGSPACE
    - Datalog (inflationary fixpoint) = PTIME
    - Datalog (partial fixpoint) = PSPACE
  - Choice of query language: tradeoff between expressiveness and optimizations

Recovery

- Transfer $100 from account #4662 to #7199:

```
X = Read(Account_1);
X.amount = X.amount - 100;
Write(Account_1, X);

Y = Read(Account_2);
Y.amount = Y.amount + 100;
Write(Account_2, Y);
```

Recovery

- Transfer $100 from account #4662 to #7199:

```
X = Read(Account_1);
X.amount = X.amount - 100;
Write(Account_1, X);

Y = Read(Account_2);
Y.amount = Y.amount + 100;
Write(Account_2, Y);
```

CRASH !

What is the problem ?

Concurrency Control

- How to overdraft your account:

User 1

User 2

```
X = Read(Account);
if (X.amount > 100)
   { dispense_money( );
      X.amount = X.amount – 100;
    }
else error(“Insufficient funds”);
```

```
X = Read(Account);
if (X.amount > 100)
   { dispense_money( );
      X.amount = X.amount – 100;
    }
else error(“Insufficient funds”);
```

What can go wrong ?

Transactions

- Recovery
- Concurrency control

ACID =
- Atomicity (= recovery)
- Consistency
- Isolation (= concurrency control)
- Durability

Take-home Message 2

- Transactions: the single most important functionality of commercial database systems
- Single-update transactions: some applications need only storage engines supporting single-update transactions
  - E.g. key-value stores
  - OLTP queries only; no decision support
  - The No-SQL movement
- Distributed systems: move away from ACID semantics to weaker isolation levels
  - This is the focus of active research today
- In 544: we will cover only traditional topics in transaction management: recovery and concurrency

Client/Server Database Architecture

- One server: stores the database
  - called DBMS or RDBMS
  - Usually a beefed-up system:
    - You can use CUBIST in this class; better: use your own computer as server
    - Large databases use a cluster of servers (parallel DBMS)

- Many clients: run apps and connect to DBMS
  - Interactive: psql (postgres), Management Studio (SQL Server)
  - Java/C++/C#/… applications
  - Connection protocol: ODBC/JDBC

Take-home Message 3

- Client/Server DBMS have higher startup-cost:
  - Need to first install/startup the server
  - Need to create logical schema, tune the physical schema
  - This is the main reason why some advanced users (scientists, researchers) stay away from RDBMS
  - After taking 544 you should no longer feel this pain
- Serverless DBMS:
  - Database system is compiled into the application's address space (as a library)
  - E.g. SQL Lite
  - Advantages: easier setup
  - Disadvantages:
    - Very limited concurrency control
    - Often these systems have only limited optimizers

SQL

- You are expected to learn SQL on your own !
  - We discuss only a few constructs in the remaining minutes of this lecture
  - Next lecture we study the relational calculus and its connection to SQL
- Resources for learning SQL:
  - The slides in this lecture
  - The textbook
  - Postgres: type \h or \?
- Start working on HW1 !

What You Should Know

- Data Manipulation Language (DML)
  – Querying: SELECT-FROM-WHERE
    - Group-by/aggregate, subqueries (especially with universal quantifiers !!), NULLs, outer-joins
  – Modifying: INSERT/DELETE/UPDATE
- Data Definition Language (DDL)
  – CREATE/ALTER/DROP
  – Constraints: will discuss these in class

Tables in SQL

Table name

Attribute names

Key

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

Data Types in SQL

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …
- Record (aka tuple)
  - Has atomic attributes
- Table (relation)
  - A set of tuples

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"

Details

- Case insensitive:

  SELECT = Select = select

  Product = product

  BUT: ‘Seattle’ ≠ ‘seattle’

- Constants:

  ‘abc’ - yes

  “abc” - no

Eliminating Duplicates

```
SELECT   DISTINCT category
FROM     Product
```

Category
Gadgets
Photography
Household

Compare to:

```
SELECT   category
FROM     Product
```

Category
Gadgets
Gadgets
Photography
Household

Ordering the Results

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

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

Ordering is ascending, unless you specify the DESC keyword.

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   DISTINCT category
FROM     Product
ORDER BY category
```

→ ?

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

→ ?

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

→ ?

Keys and Foreign Keys

Company

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

Key

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

Foreign key

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

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

Tuple Variables

Person(<u>pname</u>, address, worksfor)
Company(<u>cname</u>, address)

Which address ?

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

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

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

In Class

Product (pname, price, category, manufacturer)

Company (cname, stockPrice, country)

Find all Chinese companies that manufacture products both in the ‘toy’ category

```
SELECT   cname

FROM

WHERE
```

In Class

Product (pname, price, category, manufacturer)

Company (cname, stockPrice, country)

Find all Chinese companies that manufacture products both in the ‘electronic’ and ‘toy’ categories

```
SELECT   cname

FROM

WHERE
```

Meaning (Semantics) of SQL Queries

SELECT $a_1, a_2, \ldots, a_k$
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions

Answer = {}
for x_1 **in** R_1 **do**
    for x_2 **in** R_2 **do**
        …..
            for x_n **in** R_n **do**
                if Conditions
                    then Answer = Answer $\cup$ $\{(a_1,\ldots,a_k)\}$
return Answer

Using the Formal Semantics

What do these queries compute ?

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

Returns $R \cap S$

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

If $S \neq \varnothing$ and $T \neq \varnothing$
then returns $R \cap (S \cup T)$
else returns $\varnothing$

Joins Introduce Duplicates

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 Introduce Duplicates

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
USA
USA

Duplicates !
Remember to
add DISTINCT

Subqueries

- A *subquery* is another SQL query nested inside a larger query
- Such inner-outer queries are called *nested queries*
- A subquery may occur in:
  1. A SELECT clause
  2. A FROM clause
  3. A WHERE clause

Rule of thumb: avoid writing nested queries when possible; keep in mind that sometimes it's impossible

1. Subqueries in SELECT

Product (<u>pname</u>, price, company)

Company(<u>cname</u>, city)

For each product return the city where it is manufactured

```
SELECT X.pname, (SELECT Y.city
                 FROM Company Y
                 WHERE Y.cname=X.company)
FROM  Product X
```

What happens if the subquery returns more than one city ?

1. Subqueries in SELECT

Product (pname, price, company)

Company(cname, city)

Whenever possible, don't use a nested queries:

```
SELECT pname, (SELECT city FROM Company WHERE cname=company)
FROM  Product
```

=

```
SELECT pname, city
FROM   Product, Company
WHERE cname=company
```

We have "unnested" the query

1. Subqueries in SELECT

Product (<u>pname</u>, price, company)

Company(<u>cname</u>, city)

Compute the number of products made in each city

```
SELECT DISTINCT city, (SELECT count(*)
                       FROM Product
                       WHERE cname=company)
FROM  Company
```

Better: we can unnest by using a GROUP BY

2. Subqueries in FROM

Product (pname, price, company)

Company(cname, city)

Find all products whose prices is > 20 and < 30

```
SELECT X.city
FROM (SELECT * FROM Product AS Y WHERE Y.price > 20) AS  X
WHERE X.price < 30
```

Unnest this query !

3. Subqueries in WHERE

Product (pname, price, company)

Company(cname, city)

Existential quantifiers

Find all cities that make some products with price < 100

Using EXISTS:

```
SELECT DISTINCT  Company.city
FROM     Company
WHERE  EXISTS (SELECT *
                      FROM Product
                      WHERE company = cname and Produc.price < 100)
```

3. Subqueries in WHERE

Product (<u>pname</u>, price, company)

Company(<u>cname</u>, city)

Existential quantifiers

Find all cities that make <u>some</u> products with price < 100

Relational Calculus (a.k.a. First Order Logic)

$$\{ y \mid \exists x.\ Company(x,y) \wedge (\exists z.\ \exists p.\ Product(z,p,x) \wedge p < 100) \}$$

3. Subqueries in WHERE

Product (pname, price, company)

Company(cname, city)

Existential quantifiers

Find all cities that make some products with price < 100

Using IN

```
SELECT DISTINCT  Company.city
FROM      Company
WHERE Company.cname IN (SELECT Product.company
                        FROM Product
                        WHERE Produc.price < 100)
```

3. Subqueries in WHERE

Product (<u>pname</u>, price, company)

Company(<u>cname</u>, city)

Existential quantifiers

Find all cities that make <u>some</u> products with price < 100

Using ANY:

```
SELECT DISTINCT  Company.city
FROM     Company
WHERE 100 > ANY (SELECT price
                  FROM Product
                  WHERE company = cname)
```

3. Subqueries in WHERE

Product (<u>pname</u>, price, company)

Existential quantifiers

Company(<u>cname</u>, city)

Find all cities that make <u>some</u> products with price < 100

Now let's unnest it:

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

Existential quantifiers are easy ! ☺

3. Subqueries in WHERE

Universal quantifiers

Product (pname, price, company)

Company(cname, city)

Find all cities with companies
that make only products with price < 100

Universal quantifiers are hard ! ☹

3. Subqueries in WHERE

Product (pname, price, company)

Company(cname, city)

Universal quantifiers

Find all cities with companies
that make only products with price < 100

Relational Calculus (a.k.a. First Order Logic)

$$\{ y \mid \exists x.\ \text{Company}(x,y) \wedge (\forall z.\ \forall p.\ \text{Product}(z,p,x) \rightarrow p < 100) \}$$

3. Subqueries in WHERE

De Morgan's Laws:

$$\neg(A \land B) = \neg A \lor \neg B$$
$$\neg(A \lor B) = \neg A \land \neg B$$
$$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$$
$$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$$

$$\neg(A \rightarrow B) = A \land \neg B$$

$$\{ y \mid \exists x.\ \text{Company}(x,y) \land (\forall z.\ \forall p.\ \text{Product}(z,p,x) \rightarrow p < 100) \}$$

=

$$\{ y \mid \exists x.\ \text{Company}(x,y) \land \neg (\exists z \exists p.\ \text{Product}(z,p,x) \land p \geq 100) \}$$

=

$$\{ y \mid \exists x.\ \text{Company}(x,y)) \} \ -$$
$$\{ y \mid \exists x.\ \text{Company}(x,y) \land (\exists z \exists p.\ \text{Product}(z,p,x) \land p \geq 100) \}$$

3. Subqueries in WHERE

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

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

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

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

3. Subqueries in WHERE

Product (pname, price, company)

Company(cname, city)

Universal quantifiers

Find all cities with companies
that make only products with price < 100

Using EXISTS:

```
SELECT DISTINCT  Company.city
FROM     Company
WHERE NOT EXISTS (SELECT *
                  FROM Product
                  WHERE company = cname and Produc.price >= 100)
```

3. Subqueries in WHERE

Product (pname, price, company)

Company(cname, city)

Universal quantifiers

Find all cities with companies
that make only products with price < 100

Using ALL:

```
SELECT DISTINCT  Company.city
FROM     Company
WHERE 100 > ALL  (SELECT price
                    FROM Product
                    WHERE company = cname)
```

Question for Database Fans and their Friends

- Can we unnest the *universal quantifier* query ?

Monotone Queries

- A query Q is monotone if:
  - Whenever we add tuples to one or more of the tables…
  - … the answer to the query cannot contain fewer tuples

- **Fact**: all unnested queries are monotone
  - Proof: using the "nested for loops" semantics

- **Fact**: A query a universal quantifier is not monotone

- **Consequence**: we cannot unnest a query with a universal quantifier

Queries that must be nested

- Queries with universal quantifiers or with negation
- The drinkers-bars-beers example next
- This is a famous example from textbook on databases by Ullman

Rule of Thumb:

Non-monotone queries cannot be unnested. In particular, queries with a universal quantifier cannot be unnested

The drinkers-bars-beers example

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Challenge: write these in SQL

Find drinkers that frequent <u>some</u> bar that serves <u>some</u> beer they like.

$$x: \quad \exists y.\ \exists z.\ \text{Frequents}(x, y) \wedge \text{Serves}(y,z) \wedge \text{Likes}(x,z)$$

Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

$$x: \quad \forall y.\ \text{Frequents}(x, y) \Rightarrow (\exists z.\ \text{Serves}(y,z) \wedge \text{Likes}(x,z))$$

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they like.

$$x: \quad \exists y.\ \text{Frequents}(x, y) \wedge \forall z.(\text{Serves}(y,z) \Rightarrow \text{Likes}(x,z))$$

Find drinkers that frequent <u>only</u> bars that serves <u>only</u> beer they like.

$$x: \quad \forall y.\ \text{Frequents}(x, y) \Rightarrow \forall z.(\text{Serves}(y,z) \Rightarrow \text{Likes}(x,z))$$

Aggregation

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

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

SQL supports several aggregation operations:

sum, count, min, max, avg

Except count, all aggregations apply to a single attribute

Aggregation: Count

COUNT applies to duplicates, unless otherwise stated:

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

same as Count(*)

We probably want:

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

More Examples

Purchase(product, date, price, quantity)

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

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

What do they mean ?

Simple Aggregations

Purchase

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

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

90 (= 60+30)

Grouping and Aggregation

Purchase(product, price, quantity)

Find total quantities for all sales over $1, by product.

```
SELECT      product, Sum(quantity) AS TotalSales
FROM        Purchase
WHERE       price > 1
GROUP BY    product
```

Let's see what this means…

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUPBY

3. Compute the SELECT clause, including aggregates.

1&2. FROM-WHERE-GROUPBY

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

3. SELECT

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

Product	TotalSales
Bagel	40
Banana	20

```
SELECT      product, Sum(quantity) AS TotalSales
FROM        Purchase
WHERE       price > 1
GROUP BY    product
```

GROUP BY v.s. Nested Quereis

```
SELECT      product, Sum(quantity) AS TotalSales
FROM        Purchase
WHERE       price > 1
GROUP BY    product
```

```
SELECT DISTINCT  x.product, (SELECT Sum(y.quantity)
                             FROM     Purchase y
                             WHERE x.product = y.product
                                      AND price > 1)
                            AS TotalSales
FROM        Purchase x
WHERE       price > 1
```

Why twice ?

Another Example

```
SELECT    product,
          sum(quantity) AS SumSales
          max(price) AS MaxQuantity
FROM      Purchase
GROUP BY  product
```

What does it mean ?

Rule of thumb:

Every group in a GROUP BY is non-empty !

If we want to include empty groups in the output, then we need either a subquery, or a *left outer join* (see later)

HAVING Clause

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

```
SELECT      product, Sum(quantity)
FROM        Purchase
WHERE       price > 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

Why ?

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

Evaluation steps:

1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Advanced SQLizing

1. Unnesting Aggregates

2. Finding witnesses

Unnesting Aggregates

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

Find the number of companies in each city

```
SELECT DISTINCT city, (SELECT count(*)
                       FROM Company Y
                       WHERE X.city = Y.city)
FROM  Company X
```

Equivalent queries

```
SELECT city, count(*)
FROM   Company
GROUP BY city
```

Note: no need for DISTINCT
(DISTINCT *is the same* as GROUP BY)

Unnesting Aggregates

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

Find the number of products made in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
                         FROM Product Y, Company Z
                         WHERE Y.cname=Z.company
                             AND Z.city = X.city)
FROM  Company X
```

```
SELECT X.city, count(*)
FROM Company X, Product Y
WHERE X.cname=Y.company
GROUP BY X.city
```

They are NOT equivalent ! (WHY?)

More Unnesting

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

Wrote(login,url)

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

This is SQL by a novice

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

More Unnesting

- 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.name
HAVING      count(wrote.url) > 10
```

This is SQL by an expert

Finding Witnesses

Store(<u>sid</u>, sname)
Product(<u>pid</u>, pname, price, sid)

For each store,
find its most expensive products

Finding Witnesses

Finding the maximum price is easy…

```
SELECT Store.sid, max(Product.price)
FROM    Store, Product
WHERE  Store.sid = Product.sid
GROUP BY  Store.sid
```

But we need the *witnesses*, i.e. the products with max price

Finding Witnesses

To find the witnesses, compute the maximum price in a subquery

```
SELECT Store.sname, Product.pname
FROM Store, Product,
        (SELECT Store.sid AS sid, max(Product.price) AS p
         FROM    Store, Product
         WHERE  Store.sid = Product.sid
          GROUP BY  Store.sid, Store.sname) X
WHERE  Store.sid = Product.sid
       and Store.sid = X.sid and Product.price = X.p
```

Finding Witnesses

There is a more concise solution here:

```
SELECT Store.sname, x.pname
FROM   Store, Product x
WHERE  Store.sid = x.sid and
        x.price >=
           ALL (SELECT y.price
                FROM Product y
                WHERE Store.sid = y.sid)
```

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)
```

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

Rule in SQL: include only tuples that yield TRUE

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)

An "inner join":

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

Same as:

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

But Products that never sold will be lost !

Outerjoins

Product(<u>name</u>, category)
Purchase(prodName, store)

If we want the never-sold products, need an “outerjoin”:

```
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

Application

Compute, for each product, the total number of sales in 'September'

Product(<u>name</u>, category)

Purchase(prodName, month, store)

```
SELECT Product.name, count(*)
FROM     Product, Purchase
WHERE   Product.name = Purchase.prodName
        and  Purchase.month = 'September'
GROUP BY Product.name
```

What's wrong ?

Application

Compute, for each product, the total number of sales in 'September'

Product(name, category)

Purchase(prodName, month, store)

```
SELECT Product.name, count(store)
FROM    Product LEFT OUTER JOIN Purchase ON
                 Product.name = Purchase.prodName
           and  Purchase.month = 'September'
GROUP BY Product.name
```

Now we also get the products who sold in 0 quantity

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