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:
  – 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 postgres 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:**

<table>
<thead>
<tr>
<th>id</th>
<th>fName</th>
<th>lName</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>195428</td>
<td>Tom</td>
<td>Hanks</td>
<td>M</td>
</tr>
<tr>
<td>645947</td>
<td>Amy</td>
<td>Hanks</td>
<td>F</td>
</tr>
</tbody>
</table>

**Cast:**

<table>
<thead>
<tr>
<th>pid</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>195428</td>
<td>337166</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Movie:**

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>337166</td>
<td>Toy Story</td>
<td>1995</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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
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:

<table>
<thead>
<tr>
<th>id</th>
<th>fName</th>
<th>lName</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>Hanks</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Cast:

<table>
<thead>
<tr>
<th>pid</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

Movie:

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>1995</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Plan 1: . . . . [ in class ]

Plan 2: . . . . [ in class ]
Evaluating Tom Hanks

\[ \sigma_{\text{Name}='\text{Hanks}'} \] \( \sigma_{\text{year}=1995} \) \[ \sigma_{\text{Name}='\text{Hanks}'} \] \( \sigma_{\text{year}=1995} \)
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 = \(\mathbf{AC}^0\)
    • 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:

\[
\begin{align*}
X &= \text{Read(Account\_1)}; \\
X.\text{amount} &= X.\text{amount} - 100; \\
\text{Write(Account\_1, X)}; \\
\end{align*}
\]

\[
\begin{align*}
Y &= \text{Read(Account\_2)}; \\
Y.\text{amount} &= Y.\text{amount} + 100; \\
\text{Write(Account\_2, Y)}; \\
\end{align*}
\]
Recovery

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

\[
X = \text{Read(Account}_1); \\
X.\text{amount} = X.\text{amount} - 100; \\
\text{Write(Account}_1, X);
\]

\[
Y = \text{Read(Account}_2); \\
Y.\text{amount} = Y.\text{amount} + 100; \\
\text{Write(Account}_2, Y);
\]

What is the problem?
Concurrency Control

• How to overdraft your account:

User 1

\[ X = \text{Read(Account)}; \]
\[ \text{if} \ (X.\text{amount} \ > \ 100) \]
\[ \quad \{ \text{dispense\_money( );} \]
\[ \quad \quad X.\text{amount} = X.\text{amount} - 100; \]
\[ \quad \} \]
\[ \text{else error(“Insufficient funds”);} \]

User 2

\[ X = \text{Read(Account)}; \]
\[ \text{if} \ (X.\text{amount} \ > \ 100) \]
\[ \quad \{ \text{dispense\_money( );} \]
\[ \quad \quad X.\text{amount} = X.\text{amount} - 100; \]
\[ \quad \} \]
\[ \text{else error(“Insufficient funds”);} \]

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
<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
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

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

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

“selection”
Simple SQL Query

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

```
<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
```

“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

Compare to:

SELECT category
FROM Product

<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>
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.
SELECT DISTINCT category 
FROM Product 
ORDER BY category

SELECT Category 
FROM Product 
ORDER BY PName

SELECT DISTINCT category 
FROM Product 
ORDER BY PName

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
# Keys and Foreign Keys

## Company

<table>
<thead>
<tr>
<th>CName</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

## Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
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

SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
AND Price <= 200
Tuple Variables

Person\( (\text{pname}, \text{address}, \text{worksfor}) \)

Company\( (\text{cname}, \text{address}) \)

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT} \quad \text{pname, address} \\
\text{FROM} & \quad \text{Person, Company} \\
\text{WHERE} & \quad \text{worksfor} = \text{cname}
\end{align*}
\]

Which address?

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT} \quad \text{Person.pname, Company.address} \\
\text{FROM} & \quad \text{Person, Company} \\
\text{WHERE} & \quad \text{Person.worksfor} = \text{Company.cname}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{DISTINCT} \quad \text{x.pname, y.address} \\
\text{FROM} & \quad \text{Person AS x, Company AS y} \\
\text{WHERE} & \quad \text{x.worksfor} = \text{y.cname}
\end{align*}
\]
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 a₁, a₂, ..., aₖ
FROM R₁ AS x₁, R₂ AS x₂, ..., Rₙ AS xₙ
WHERE Conditions

Answer = {}
for x₁ in R₁ do
  for x₂ in R₂ do
    ..... 
    for xₙ in Rₙ do
      if Conditions 
        then Answer = Answer ∪ {(a₁,...,aₖ)}
return Answer
```
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 \( S \neq \emptyset \) and \( T \neq \emptyset \)
then returns \( R \cap (S \cup T) \)
else returns \( \emptyset \)
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

<table>
<thead>
<tr>
<th>Name</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

### Company

<table>
<thead>
<tr>
<th>Cname</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

### SQL Query

```sql
SELECT Country
FROM Product, Company
WHERE Manufacturer=CName AND Category='Gadgets'
```

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

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

Existential quantifiers

Find all cities that make some products with price < 100

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

\{ y | \exists x. Company(x,y) \land (\exists z. \exists p. Product(z,p,x) \land p < 100) \}
3. Subqueries in WHERE

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

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 Product.price < 100)
```
3. Subqueries in WHERE

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

Find all cities that make **some** 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 (pname, price, company)  
Company (cname, city)

Find all cities that make some 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 😊

Dan Suciu -- 544, Winter 2011
3. Subqueries in WHERE

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)  Universal quantifiers
Company(cname, city)

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

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

{ y | \exists x. Company(x,y) \land (\forall z. \forall p. Product(z,p,x) \Rightarrow p < 100) }
3. Subqueries in WHERE

De Morgan’s Laws:

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

\[
\{ y \mid \exists x. \text{Company}(x,y) \land (\forall z. \forall p. \text{Product}(z,p,x) \rightarrow p < 100) \}\ = \\
\{ y \mid \exists x. \text{Company}(x,y) \land \neg (\exists z \exists p. \text{Product}(z,p,x) \land p \geq 100) \}\ = \\
\{ y \mid \exists x. \text{Company}(x,y)) \} - \\
\{ y \mid \exists x. \text{Company}(x,y) \land (\exists z \exists p. \text{Product}(z,p,x) \land p \geq 100) \}\]
3. Subqueries in WHERE

1. Find the other companies: i.e. s.t. some product $\geq$ 100

   ```sql
   SELECT DISTINCT Company.city
   FROM Company
   WHERE Company.cname IN (SELECT Product.company
                              FROM Product
                              WHERE Product.price $\geq$ 100)
   ```

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

   ```sql
   SELECT DISTINCT Company.city
   FROM Company
   WHERE Company.cname NOT IN (SELECT Product.company
                                 FROM Product
                                 WHERE Product.price $\geq$ 100)
   ```
3. Subqueries in WHERE

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

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)  Universal quantifiers
Company (cname, city)

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

Find drinkers that frequent some bar that serves some beer they like.

\[ x: \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent only bars that serves some beer they like.

\[ x: \forall y. \text{Frequents}(x, y) \Rightarrow (\exists z. \text{Serves}(y, z) \land \text{Likes}(x, z)) \]

Find drinkers that frequent some bar that serves only beers they like.

\[ x: \exists y. \text{Frequents}(x, y) \land \forall z. (\text{Serves}(y, z) \Rightarrow \text{Likes}(x, z)) \]

Find drinkers that frequent only bars that serves only beer they like.

\[ x: \forall y. \text{Frequents}(x, y) \Rightarrow \forall z. (\text{Serves}(y, z) \Rightarrow \text{Likes}(x, z)) \]
Aggregation

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)

\[
\text{SELECT Sum(price * quantity) FROM Purchase}
\]

\[
\text{SELECT Sum(price * quantity) FROM Purchase WHERE product = 'bagel'}
\]

What do they mean?
Simple Aggregations

**Purchase**

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

**Query:**

```
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

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>
### 3. SELECT

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

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>40</td>
</tr>
<tr>
<td>Banana</td>
<td>20</td>
</tr>
</tbody>
</table>
GROUP BY v.s. Nested Queries

```
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

```sql
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

\[
\begin{align*}
\text{SELECT} & \quad S \\
\text{FROM} & \quad R_1, \ldots, R_n \\
\text{WHERE} & \quad C_1 \\
\text{GROUP BY} & \quad a_1, \ldots, a_k \\
\text{HAVING} & \quad C_2
\end{align*}
\]

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

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

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

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

Equivalent queries

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(login, name)
Wrote(login, url)

• Find authors who wrote ≥ 10 documents:

• Attempt 1: with nested queries

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

This is SQL by a novice
More Unnesting

• Find all authors who wrote at least 10 documents:

• Attempt 2: SQL style (with GROUP BY)

```sql
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

Dan Suciu -- 544, Winter 2011
Finding Witnesses

Store(sid, sname)
Product(pid, 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:

```sql
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 $x = \text{NULL}$ then $4 \times (3-x)/7$ is still $\text{NULL}$

• If $x = \text{NULL}$ then $x = 'Joe'$ is \text{UNKNOWN}

• In SQL there are three boolean values:
  \begin{align*}
    \text{FALSE} &= 0 \\
    \text{UNKNOWN} &= 0.5 \\
    \text{TRUE} &= 1
  \end{align*}
Null Values

- \( C_1 \text{ AND } C_2 = \min(C_1, C_2) \)
- \( C_1 \text{ OR } C_2 = \max(C_1, C_2) \)
- \( \text{NOT } C_1 = 1 - C_1 \)

Rule in SQL: include only tuples that yield TRUE

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

E.g.
- age=20
- heighth=NULL
- weight=200
Null Values

Unexpected behavior:

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

Some Persons are not included!
Null Values

Can test for NULL explicitly:

– x IS NULL
– x IS NOT NULL

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

Now it includes all Persons
Outerjoins

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

An “inner join”:

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

Same as:

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

But Products that never sold will be lost!
Outerjoins

Product(name, 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

### Purchase

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
<tr>
<td>OneClick</td>
<td>NULL</td>
</tr>
</tbody>
</table>
Application

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

Product(name, 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