Introduction to Database Systems
CSEP544

Lecture #1
January 5, 2007
About Me

Dan Suciu:
• Bell Labs, AT&T Labs, UW in 2000

Research:
• Past: XML and semi-structured data:
  – Query language: XML-QL (later XQuery)
  – Compressor: XMill
  – Theory: XPath containment, XML typechecking
• Present: Probabilistic databases: MystiQ
Staff

• Instructor: Dan Suciu
  – Allen, Room 662, suciu@cs.washington.edu
  Office hours: Tuesdays, 5:30 (appointment strongly recommended)
  – Next week: away, chairing ICDT’2007

• TAs:
  – Bao Nguyen Nguyen
Communications

• Web page: 
  http://www.cs.washington.edu/p544/  □□
  – Lectures will be available here
  – Homeworks will be posted here (HW1 is posted)
  – The project description will be here

• Mailing list:
  – Announcements, group discussions
  – Please subscribe
Textbook(s)

Main textbook, available at the bookstore:

• *Database Management Systems*
  Ramakrishnan and Gehrke

Also recommended:

• *Database Systems: The Complete Book,*
  Garcia-Molina, Ullman, Widom

Gives colloquium talk on Feb.8, 3:30pm
Other Texts

Available at the Engineering Library (not on reserve):

• *XQuery from the Experts*, Katz, Ed.
• *Foundations of Databases*, Abiteboul, Hull, Vianu
• *Data on the Web*, Abiteboul, Buneman, Suciu
Outline of Today’s Lecture

1. Overview of DBMS, Course outline

2. Assignment 1, Homework 1, Project phase 1

3. SQL
Database

What is a database?

Give examples of databases
Database

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 2004 www.computerworld.com

- IMB: 35% market with $2.5BN in sales
- Oracle: 33% market with $2.3BN in sales
- Microsoft: 19% market with $1.3BN 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, …
## Tables

### Directors:

<table>
<thead>
<tr>
<th>id</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>15901</td>
<td>Francis Ford</td>
<td>Coppola</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Movie_Directors:

<table>
<thead>
<tr>
<th>id</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>15901</td>
<td>130128</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Movies:

<table>
<thead>
<tr>
<th>mid</th>
<th>Title</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>130128</td>
<td>The Godfather</td>
<td>1972</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

|         |             |      |
What the Database Systems Does

1. Create/store large datasets
2. Search/query/update
3. Change the structure
4. Concurrent access to many user
5. Recover from crashes
6. Security
Possible Organizations

- Files
- Spreadsheets
- DBMS
1. Create/store Large Datasets

- Files: Yes, but…
- Spreadsheets: Not really…
- DBMS: Yes
2. Search/Query/Update

- Files
  - Simple queries (grep);
  - Updates are difficult

- Spreadsheets
  - Simple queries;
  - Simple updates

- DBMS
  - All

Updates: generally OK
3. Change the Structure

Add \textit{Address} to each Actor

- Files
  - Very hard

- Spreadsheets
  - Yes

- DBMS
  - Yes
4. Concurrent Access

Multiple users access/update the data concurrently

- What can go wrong?
- How do we protect against that in OS?
- This is insufficient in databases; why?

Lost updates; inconsistent reads,…

A logical action consists of multiple updates

locks
5. Recover from crashes

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

  \[
  \begin{align*}
  X &= \text{Read(Account, #4662)}; \\
  X.\text{amount} &= X.\text{amount} - 100; \\
  \text{Write(Account, #4662, X)}; \\
  \\
  Y &= \text{Read(Account, #7199)}; \\
  Y.\text{amount} &= Y.\text{amount} + 100; \\
  \text{Write(Account, #7199, Y)};
  \end{align*}
  \]

What is the problem?
6. Security

- Files
- Spreadsheets
- DBMS

File-level access control

Same [?]

Table/attribute-level access control
Enters a DMBS

“Two tier system” or “client-server”

Data files

Database server (someone else’s C program)

Applications

connection

(ODBC, JDBC)
# Data Independence

## Logical view

### Directors:

<table>
<thead>
<tr>
<th>id</th>
<th>fName</th>
<th>lName</th>
</tr>
</thead>
<tbody>
<tr>
<td>15901</td>
<td>Francis Ford</td>
<td>Coppola</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Movie_Directors:

<table>
<thead>
<tr>
<th>id</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>15901</td>
<td>130128</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

### Movies:

<table>
<thead>
<tr>
<th>mid</th>
<th>Title</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>130128</td>
<td>The Godfather</td>
<td>1972</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

## Physical view

- Director file
- Movie_title_index file
- Director_fname_index file
- Movie file
What the Database Systems Does

1. Create/store large datasets
2. Search/query/update
3. Change the structure
4. Concurrent access to many user
5. Recover from crashes
6. Security

SQL DML

Transactions
ACID

Grant, Revoke, Roles
Course Outline - TENTATIVE !!

1. January 5: SQL
4. January 30: XML/XPath/XQuery
5. February 6: Transactions
6. February 13: Database storage, indexes
7. February 20: Physical operators, optimization
8. February 27: Statistics, Database tuning
9. March 6: Advanced topics
Grading

• Homework: 35%
• Project: 35%
• Final: 30%
Reading Assignment

• Reading assignment for Tuesday, Jan 16
  – Introduction from SQL for Web Nerds,

• This is a one-time assignment, no grading, BUT
  very instructive and lots of fun reading
Homework 1

- Homework 1:
  - SQL Queries
  - Due Tuesday, January 16
  - It is posted already!

- Homework 2:
  - Conceptual design: E/R diagrams, Normal Forms
  - Due Tuesday, January 30

- Homework 3:
  - XML/Xquery
  - Due Tuesday, February 13

- Homework 4:
  - Transactions: concurrency control and recovery
  - Due Tuesday, February 27
The Project:
Boutique Online Store

• Phase 1:
  – Design a Database Schema, Build Related Data Logic
  – Due January 23

• Phase 2:
  – Import data, Web Inventory Data Logic
  – Due February 6

• Phase 3:
  – Checkout Logic
  – Due February 20

• Phase 4:
  – Database Tuning
  – Due March 6
Project

SQL Server, C#, ASP.NET
• Supported
• Will provide starter code in C#, ASP.NET
• The import data is in SQL/XML on SQL Server

Alternative technologies: MySQL, postgres, PHPs
• Not supported (you are on your own)
• Worry about the SQL/XML part…
Accessing SQL Server

**SQL Server Management Studio**
- **Server Type** = Database Engine
- **Server Name** = IPROJSRV
- **Authentication** = SQL Server Authentication
  - Login = your UW email address (*not* the CSE email)
  - Password = 12345

Change your password !!

Then play with IMDB
Today’s Lecture: SQL

• Chapters 5.1 - 5.5

• If we don’t finish today please read the slides at home: you need this material for the Homework due next time.
SQL Introduction

Standard language for querying and manipulating data

Structured Query Language

Many standards out there:
• ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
• Vendors support various subsets: watch for fun discussions in class!
SQL

- **Data Definition Language (DDL)**
  - Create/alter/delete tables and their attributes
  - Following lectures...
- **Data Manipulation Language (DML)**
  - Query one or more tables – discussed next!
  - Insert/delete/modify tuples in tables
### Tables in SQL

<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>
Tables Explained

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

Product(PName, Price, Category, Manufacturer)

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

Product(PName, Price, Category, Manufacturer)
Data Types in SQL

• Atomic types:
  – Characters: CHAR(20), VARCHAR(50)
  – Numbers: INT, BIGINT, SMALLINT, FLOAT
  – Others: MONEY, DATETIME, …

• Every attribute must have an atomic type
  – Hence tables are flat
  – Why?
Tables Explained

• A tuple = a record
  – Restriction: all attributes are of atomic type

• A table = a set of tuples
  – Like a list…
  – …but it is unordered:
    no first(), no next(), no last().
SQL Query

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

```
SELECT <attributes> FROM <one or more relations>
WHERE <conditions>
```
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

### SQL Query

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

### Data Table

<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>

### Diagram

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

### Analysis

- **“selection” and “projection”**
Notation

Input Schema
Product(PName, Price, Category, Manufacturer)

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

Answer(PName, Price, Manufacturer)

Output Schema
Details

- Case insensitive:
  - Same: SELECT Select select
  - Same: Product product
  - Different: ‘Seattle’ ‘seattle’

- Constants:
  - ‘abc’ - yes
  - “abc” - no
The **LIKE** operator

```
SELECT  *  
FROM    Products
WHERE   PName LIKE '%%gizmo%%'
```

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

SELECT DISTINCT category
FROM Product

Compare to:

SELECT category
FROM Product
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, etc.

Ordering is ascending, unless you specify the DESC keyword.
<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>

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

<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.

\[
\text{SELECT PName, Price FROM Product, Company WHERE Manufacturer=CName AND Country='Japan' AND Price <= 200}
\]
Joins

<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>

<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>

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

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
</tr>
</tbody>
</table>
More Joins

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

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

```sql
SELECT cname
FROM
WHERE
```
A Subtlety about Joins

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

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

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

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

What is the problem? What’s the solution?
Tuple Variables

Person(pname, address, worksfor)
Company(cname, 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
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
```
An Unintuitive Query

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

What does it compute?

Computes $R \cap (S \cup T)$ most of the time

When does it not compute $R \cap (S \cup T)$?
Subqueries Returning Relations

Company(name, city)
Product(pname, maker)
Purchase(id, product, buyer)

Return cities where one can find companies that manufacture products bought by Joe Blow

```
SELECT Company.city
FROM Company
WHERE Company.name IN
  (SELECT Product.maker
   FROM Purchase , Product
   WHERE Product.pname=Purchase.product
   AND Purchase .buyer = 'Joe Blow');
```
Subqueries Returning Relations

Is it equivalent to this?

```sql
SELECT Company.city
FROM Company, Product, Purchase
WHERE Company.name = Product.maker
  AND Product.pname = Purchase.product
  AND Purchase.buyer = 'Joe Blow'
```
Removing Duplicates

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

```
SELECT DISTINCT Company.city
FROM   Company, Product, Purchase
WHERE  Company.name = Product.maker
       AND Product.pname = Purchase.product
       AND Purchase.buyer = 'Joe Blow'
```

Now they are equivalent
Subqueries Returning Relations

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

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

```
SELECT name
FROM Product
WHERE price > ALL (SELECT price
FROM Product
WHERE maker='Gizmo-Works')
```
Question for Database Fans and their Friends

• Can we express this query as a single SELECT-FROM-WHERE query, without subqueries?
Monotone Queries

Let Q be a query over tables R, S, T, …; denote its answer with Q(R, S, T, …).

**Definition** Q is called monotone if:
\[ \forall R \subseteq R', S \subseteq S', \ldots \Rightarrow Q(R, S, \ldots) \subseteq Q(R', S', \ldots) \]

**Theorem** Every select-from-where query is monotone

**Observation** The **ALL** query on previous slide is not monotone
Correlated Queries

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

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

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

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

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

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

Very powerful! Also much harder to optimize.
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:

\[
\text{SELECT} \quad \text{Count(category)} \\
\text{FROM} \quad \text{Product} \\
\text{WHERE} \quad \text{year} > 1995
\]

same as \(\text{Count(\ast)}\)

We probably want:

\[
\text{SELECT} \quad \text{Count(DISTINCT category)} \\
\text{FROM} \quad \text{Product} \\
\text{WHERE} \quad \text{year} > 1995
\]
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

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
</tbody>
</table>

```sql
SELECT Sum(price * quantity) FROM Purchase WHERE product = 'bagel'  
50 (= 20+30)
```
Grouping and Aggregation

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product.

```
SELECT  product, Sum(price*quantity) AS TotalSales
FROM    Purchase
WHERE   date > '10/1/2005'
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: grouped attributes and aggregates.
### 1&2. FROM-WHERE-GROUPBY

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1.00</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1.00</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>0.50</td>
<td>10</td>
</tr>
</tbody>
</table>
3. SELECT

<table>
<thead>
<tr>
<th>Product</th>
<th>Date</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>10/21</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>10/25</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>10/3</td>
<td>0.5</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>10/10</td>
<td>1</td>
<td>10</td>
</tr>
</tbody>
</table>

SELECT product, Sum(price*quantity) AS TotalSales FROM Purchase WHERE date > '10/1/2005' GROUP BY product

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>
GROUP BY v.s. Nested Queries

```
SELECT product, Sum(price*quantity) AS TotalSales
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
```

```
SELECT DISTINCT x.product, (SELECT Sum(y.price*y.quantity)
                           FROM Purchase y
                           WHERE x.product = y.product
                           AND y.date > '10/1/2005')
                           AS TotalSales
FROM Purchase x
WHERE x.date > '10/1/2005'
```
Another Example

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

What does it mean?
HAVING Clause

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

```sql
SELECT product, Sum(price * quantity)
FROM Purchase
WHERE date > '10/1/2005'
GROUP BY product
HAVING Sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
General form of Grouping and Aggregation

SELECT  S
FROM    R_1, ..., R_n
WHERE   C_1
GROUP BY a_1, ..., a_k
HAVING  C_2

S = may contain attributes a_1, ..., a_k and/or any aggregates but NO OTHER ATTRIBUTES
C_1 = is any condition on the attributes in R_1, ..., R_n
C_2 = 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 a₁,..,aₖ
3. Apply condition C₂ to each group (may have aggregates)
4. Compute aggregates in S and return the result
Advanced SQLizing

1. Getting around INTERSECT and EXCEPT

2. Quantifiers

3. Aggregation v.s. subqueries

4. Two examples (study at home)
1. INTERSECT and EXCEPT:

Intersect:

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

Except:

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

If R, S have no duplicates, then can write without subqueries (HOW?)

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

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

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

Find all companies that make some products with price < 100

\[
\text{SELECT DISTINCT Company.cname FROM Company, Product WHERE Company.cname = Product.company and Product.price < 100}
\]

Existential: easy 😊
2. Quantifiers

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

Find all companies that make only products with price < 100

same as:

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

Universal: hard! 😞
2. Quantifiers

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

```sql
SELECT DISTINCT Company.cname
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.cname
FROM Company
WHERE Company.cname NOT IN (SELECT Product.company
FROM Product
WHERE Product.price $\geq$ 100)
```
3. Group-by v.s. Nested Query

Author(login,name)
Wrote(login,url)

• Find authors who wrote $\geq 10$ documents:
• Attempt 1: with nested queries

```
SELECT DISTINCT Author.name
FROM Author
WHERE count(SELECT Wrote.url
FROM Wrote
WHERE Author.login=Wrote.login) > 10
```
3. Group-by v.s. Nested Query

• 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
```

No need for **DISTINCT**: automatically from **GROUP BY**
3. Group-by v.s. Nested Query

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

Find authors with vocabulary \( \geq 10000 \) words:

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

Store(sid, sname)
Product(pid, pname, price, sid)

Find all stores that sell *only* products with price > 100

same as:

Find all stores s.t. all their products have price > 100
SELECT Store.name
FROM Store, Product
WHERE Store.sid = Product.sid
GROUP BY Store.sid, Store.name
HAVING 100 < min(Product.price)

Almost equivalent…

SELECT Store.name
FROM Store
WHERE
  100 < ALL (SELECT Product.price
              FROM product
              WHERE Store.sid = Product.sid)

SELECT Store.name
FROM Store
WHERE
  Store.sid NOT IN
    (SELECT Product.sid
     FROM Product
     WHERE Product.price <= 100)
Two Examples

Store(sid, sname)
Product(pid, pname, price, sid)

For each store,
find its most expensive product
Two Examples

This is easy but doesn’t do what we want:

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

Better:

```
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)
```
Two Examples

Finally, choose some pid arbitrarily, if there are many with highest price:

```
SELECT Store.sname, max(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) 
GROUP BY Store.sname
```
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 \( 4*(3-x)/7 \) is still \( \text{NULL} \)

• If \( x = \text{NULL} \) then \( x=\text{“Joe”} \) is \( \text{UNKNOWN} \)

• In SQL there are three boolean values:
  
<table>
<thead>
<tr>
<th>Boolean Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>0.5</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
</tr>
</tbody>
</table>
Null Values

- C1 AND C2 = \min(C1, C2)
- C1 OR C2 = \max(C1, C2)
- \text{NOT } C1 = 1 - C1

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

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

Unexpected behavior:

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

Some Persons are not included!
Null Values

Can test for NULL explicitly:
  – x IS NULL
  – x IS NOT NULL

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

Now it includes all Persons
Outerjoins

Explicit joins in SQL = “inner joins”:

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!
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

<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>

### Purchase (with NULL values)

<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>

99
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(*)
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
Modifying the Database

Three kinds of modifications

• Insertions
• Deletions
• Updates

Sometimes they are all called “updates”
Insertions

General form:

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

Example: Insert a new purchase to the database:

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

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

\[
\text{INSERT INTO PRODUCT(name)} \\
\quad \text{SELECT DISTINCT Purchase.product} \\
\quad \text{FROM Purchase} \\
\quad \text{WHERE Purchase.date > "10/26/01"}
\]

The query replaces the VALUES keyword. Here we insert many tuples into PRODUCT
Insertion: an Example

Product\(\text{name}, \text{listPrice}, \text{category}\)
Purchase(\text{prodName}, \text{buyerName}, \text{price})

\text{prodName} \text{ is foreign key in } \text{Product.name}

Suppose database got corrupted and we need to fix it:

Task: insert in \text{Product} all \text{prodNames} from \text{Purchase}
Insertion: an Example

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

<table>
<thead>
<tr>
<th>name</th>
<th>listPrice</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>gizmo</td>
<td>100</td>
<td>Gadgets</td>
</tr>
<tr>
<td>camera</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Insertion: an Example

```
INSERT INTO Product(name, listPrice)

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

<table>
<thead>
<tr>
<th>name</th>
<th>listPrice</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>gizmo</td>
<td>100</td>
<td>Gadgets</td>
</tr>
<tr>
<td>camera</td>
<td>200</td>
<td>-</td>
</tr>
<tr>
<td>camera ??</td>
<td>225 ??</td>
<td>-</td>
</tr>
</tbody>
</table>

→ 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 seen 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
Creating Tables

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>VARCHAR(30)</td>
</tr>
<tr>
<td>social-security-number</td>
<td>INT</td>
</tr>
<tr>
<td>age</td>
<td>SHORTINT</td>
</tr>
<tr>
<td>city</td>
<td>VARCHAR(30)</td>
</tr>
<tr>
<td>gender</td>
<td>BIT(1)</td>
</tr>
<tr>
<td>Birthdate</td>
<td>DATE</td>
</tr>
</tbody>
</table>

```sql
CREATE TABLE Person(
    name VARCHAR(30),
    social-security-number INT,
    age SHORTINT,
    city VARCHAR(30),
    gender BIT(1),
    Birthdate DATE
);
```
Deleting or Modifying a Table

Deleting:
Example: 

```
DROP Person;
```

Exercise with care !!

Altering: (adding or removing an attribute).

Example:

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

```
ALTER TABLE Person
DROP age;
```

What happens when you make changes to the schema?
Default Values

Specifying default values:

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

The default of defaults: NULL
Indexes

REALLY important to speed up query processing time.

Suppose we have a relation

Person (name, age, city)

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

Sequential scan of the file Person may take long
Indexes

- Create an index on name:

B+ trees have fan-out of 100s: max 4 levels!
Will discuss in the second half of this course
Creating Indexes

Syntax:

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

Indexes can be useful in range queries too:

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

B+ trees help in:

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

Why not create indexes on everything?
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"
```

and even in:
```
SELECT * FROM Person WHERE age = 55
```

But not in:
```
SELECT * FROM Person WHERE city = "Seattle"
```
The Index Selection Problem

• Why not build an index on every attribute? On every pair of attributes? Etc.? 

• The index selection problem is hard: balance the query cost v.s. the update cost, in a large application workload