Staff

Instructor: Prof. Dan Suciu:

- Bell Labs, AT&T Labs, UW, MSR
- Research interests:
  - Semi-structured data (XML): XML-QL, XMill (XML compressor), XPath containment
  - Probabilistic Databases
  - Database Security and Privacy
- CSE 662, suciu@cs.washington.edu, Office hours by email appointments (any day OK except Tuesday)

TA: Bhushan Mandhani
Communications

  - Lectures are here
  - The homework assignments are here

- **Mailing list:**
  - Announcements, group discussions
  - Please subscribe
Textbook(s)

Main textbook:
• *Database Management Systems*, Ramakrishnan and Gehrke

Second textbook:

Most important: COME TO CLASS! ASK QUESTIONS!
Other Texts

• *Fundamentals of Database Systems*, Elmasri, Navathe

• *XQuery from the Experts*, Katz, Ed.

• *Foundations of Databases*, Abiteboul, Hull, Vianu

• *Data on the Web*, Abiteboul, Buneman, Suciu
Course Format

• Lectures Tuesdays, 6:30-9:20

• 7 Homework Assignments

• Final
Grading

• Homework Assignments: 70%

• Final: 30%
7 Homework Assignments

1. SQL (already posted)
2. Conceptual Design (already posted)
3. SQL in Java
4. Transactions
5. Database tuning
6. Query optimization
7. XQuery

Due: Tuesdays, every week, by email to Bhushan
Final

• Need to reschedule the official date (June 11)

• Proposed date for the final:

  TUESDAY, JUNE 9, 2009, 6:30-8:20pm

• If you can’t make it, let me know by email; I’d like to make the date official next week
Outline of Today’s Lecture

1. Overview of DBMS

2. Course content

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

### 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>
<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>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>
<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
SQL

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

This query has selections and joins
How Can We Evaluate the Query?

<table>
<thead>
<tr>
<th>Actor:</th>
<th>Cast:</th>
<th>Movie:</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Table" /></td>
<td><img src="image" alt="Table" /></td>
<td><img src="image" alt="Table" /></td>
</tr>
</tbody>
</table>

Plan 1: . . . [ in class ]

Plan 2: . . . [ in class ]
Evaluating Tom Hanks
What Functionality Should a DBMS Support?

• [in class]
What Functionality Should a DBMS Support?

1. Data independence
2. Efficient data access
3. Data integrity and security
4. Concurrent access
5. Crash recovery
1. Data Independence

• Separation between:
  – Physical representation of the data
  – Logical view of the data

• The physical rep may change to improve efficiency (add/drop index, etc)

• Applications not affected: they see only the logical view
2. Efficient Data Access

- Physical data storage: indexes, data clustering
- Query processing: efficient algorithms for accessing/processing the data
- Query optimization: choosing between alternative, equivalent plans

Lectures 6, 7
3. Data Integrity and Security

- Integrity: enforce application constraints during database updates
- Security: access control to the data
4. Concurrency Control

User 1:

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);

User 2:

X = Read(Account#2);
X.amount = X.amount - 30;
Write(Account#2, X);

Y = Read(Account#3);
Y.amount = Y.amount + 30;
Write(Account#3, Y);

What can go wrong?

Lecture 4
5. Recovery from Crashes

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 can go wrong?
Data Management Beyond DBMS

• Other data formats:
  – Semistructured data: XML
  – XPath/XQuery

• Large scale data processing
  – Stream processing
  – Advanced hashing techniques (min-hashes, LSH)
  – Sampling

Lectures 8, 9, 10
(An Example)

Quiz:
• Alice sends Bob in random order all the numbers 1, 2, 3, …, 100000000000000000000
• She does not repeat any number
• But she misses exactly one number !
• Help Bob find out which one is missing !

Solved it ? Try this:
• As above, but Alice misses exactly ten numbers !
Lectures

1. SQL (today)
2. Database design, Normal Forms
3. Constraints, Views, Security
4. Transactions (recovery)
5. Transactions (concurrency control)
6. Data storage, indexes, physical tuning
7. Query execution and optimization
8. XML/Xpah/Xquery
9. -- 10. Advanced topics
Homeworks

1. SQL [SQL Server] 4/14
2. Conceptual Design 4/21
3. SQL App [SQL Server + Postgres] 4/28
4. Transactions 5/5
5. Database Tuning [Postgres] 5/19
6. Optimizations 5/26
7. XQuery [Galax] 6/2
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 = [login]_P544  Change it!

[See tunneling, MSDNAA]
Then play with IMDB, start working on HW 1
Today: SQL!

- Datatypes in SQL
- Simple Queries in SQL
- Joins
- Subqueries
- Aggregates
- Nulls
- Outer joins
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>
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

<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>

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

"selection"
**Simple SQL Query**

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

### 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>

“selection” and “projection”
Details

• Case insensitive:
  – SELECT = Select = select
  – Product = product
  – BUT: ‘Seattle’ ≠ ‘seattle’

• Constants:
  – ‘abc’ - yes
  – “abc” - no
Eliminating Duplicates

```sql
SELECT DISTINCT category
FROM Product
```

Compare to:

```sql
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, 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 over $100 manufactured in Japan; return their names and prices.

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
AND Price >= 100
```
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 >= 100
In Class

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

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

SELECT  cname
FROM
WHERE
In Class

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

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

```
SELECT  cname
FROM
WHERE
```
In Class

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

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

SELECT z.cname
FROM Product x, Product y, Company z
WHERE x.cname=z.cname and y.cname=z.cname and x.category='electronic' and y.category='toy'
Meaning (Semantics) of SQL Queries

```
SELECT a_1, a_2, ..., 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 ∪ {(a_1,...,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**

Returns $R \cap (S \cup T)$
If $S \neq \phi$ and $T \neq \phi$
Subqueries

A subquery (aka *nested* query) may occur in:
1. A SELECT clause
2. A FROM clause
3. A WHERE clause

**Rule of thumb**: avoid nested queries when possible; sometimes cannot avoid them
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:

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

We have “unnested” the query.

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

```sql
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

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

Existential quantifiers are easy ! 😊
3. Subqueries in WHERE

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

Find all cities with companies that make only products with price < 100
same as:
Find all cities with companies where all products have price < 100

Universal quantifiers are hard! 😞
3. Subqueries in WHERE

1. Find the other companies: i.e. s.t. some product ≥ 100

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

2. Now, find all companies s.t. all 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 \textbf{WHERE}

\begin{itemize}
\item \textbf{Product} (\textit{pname, price, company})
\item \textbf{Company} (\textit{cname, city})
\end{itemize}

Find all cities with companies that make \textbf{only} products with price $< 100$

Using \textbf{EXISTS}:

\begin{verbatim}
SELECT DISTINCT Company.city 
FROM Company 
WHERE NOT EXISTS (SELECT * 
    FROM Product 
    WHERE company = cname and Product.price $\geq$ 100)
\end{verbatim}
3. Subqueries in WHERE

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

Find all cities that make some 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
Non-monotone queries cannot be unnested. In particular, queries with universal 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 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.

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

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

COUNT applies to duplicates, unless otherwise stated:

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

Almost the same as Count(*):
- count(category) does not count any category = NULL

We probably want:

```
SELECT Count(DISTINCT category)
FROM Product
WHERE year > 1995
```
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: grouped attributes and 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 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       y.price > 1)
                        AS TotalSales
FROM    Purchase x
WHERE   x.price > 1
```

Why twice?
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)

```
SELECT R.A, count(*)
FROM R
WHERE R.B < 55
GROUP BY R.A
```

Always > 0 (Why?)
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

\[
\text{SELECT } S \\
\text{FROM } R_1, \ldots, R_n \\
\text{WHERE } C1 \\
\text{GROUP BY } a_1, \ldots, a_k \\
\text{HAVING } C2
\]

S = may contain attributes $a_1, \ldots, a_k$ and/or any aggregates but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in $R_1, \ldots, R_n$

C2 = is any condition on aggregate expressions
General form of Grouping and Aggregation

Evaluation steps:
Evaluate FROM-WHERE, apply condition C1
Group by the attributes a₁,...,aₖ
Apply condition C2 to each group (may have aggregates)
Compute aggregates in S and return the result

```
SELECT  S
FROM    R₁,...,Rₙ
WHERE   C₁
GROUP BY a₁,...,aₖ
HAVING  C₂
```
Advanced SQLizing

1. INTERSECT and EXCEPT

2. Unnesting Aggregates

3. Finding witnesses
INTERSECT and EXCEPT: not in some DBMS

INTERSECT and EXCEPT:

\[
\begin{align*}
\text{(SELECT R.A, R.B FROM R) INTERSECT (SELECT S.A, S.B FROM S)} & = \\
\text{(SELECT R.A, R.B FROM R) EXCEPT (SELECT S.A, S.B FROM S)} & = \\
\text{SELECT R.A, R.B FROM R WHERE EXISTS (SELECT * FROM S WHERE R.A=S.A and R.B=S.B)} & = \\
\text{SELECT R.A, R.B FROM R WHERE NOT EXISTS (SELECT * FROM S WHERE R.A=S.A and R.B=S.B)} & =
\end{align*}
\]

Can unnest. How?
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
```
Unnesting Aggregates

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

Find the number of products made in each city

\[
\text{SELECT DISTINCT } X.\text{city}, (\text{SELECT count(*) FROM Product Y, Company Z WHERE Y.cname= } X.\text{company AND Z.city = } X.\text{city})
\]

\[
\text{FROM Company X}
\]

They are NOT equivalent! (WHY?)
More Unnesting

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

• Find authors who wrote \( \geq 10 \) documents;
• Attempt 1: with nested queries

```sql
SELECT DISTINCT Author.name
FROM Author
WHERE (SELECT count(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)

```
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(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) 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:

\[
\begin{align*}
\text{SELECT} & \quad \text{Store.sname, x.pname} \\
\text{FROM} & \quad \text{Store, Product x} \\
\text{WHERE} & \quad \text{Store.sid = x.sid and} \\
& \quad x.\text{price} \geq \ \\
& \quad \text{ALL (SELECT y.price} \\
& \quad \text{FROM Product y} \\
& \quad \text{WHERE Store.sid = y.sid)}
\end{align*}
\]
NULLS in SQL

• Whenever we don’t have a value, we can put a NULL
• Can mean many things:
  – Value does not exist
  – 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)

Rule in SQL: include only tuples that yield TRUE

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

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(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

Need to use attribute to get correct zero count
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