SQL

Structured Query Language
SQL

• Data Definition Language (DDL)
  – Create/alter/delete tables and their attributes
  – We won’t cover this ...

• 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>

#### Attribute names
- **Product**
- **Table name**
- **Tuples or rows**
Tables Explained

• The *schema* of a table is the table name and its attributes:
  Product(PName, Price, Category, Manfacturer)

• A *key* is an attribute whose values are unique; we underline a key
  Product(PName, Price, Category, Manfacturer)
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'
```

### 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”
Simple SQL Query

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

“selection” and “projection”
Notation

\[
\text{Input Schema:} \quad \text{Product}(\text{PName}, \text{Price}, \text{Category}, \text{Manufacturer})
\]

\[
\begin{align*}
\text{SELECT} & \quad \text{PName, Price, Manufacturer} \\
\text{FROM} & \quad \text{Product} \\
\text{WHERE} & \quad \text{Price} > 100
\end{align*}
\]

\[
\text{Output Schema:} \quad \text{Answer}(\text{PName}, \text{Price}, \text{Manufacturer})
\]
Details

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

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

```sql
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

<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>
Ordering the Results

```sql
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.
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.

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

<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>

SingleTouch $149.99
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

```
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’
```

Unexpected duplicates
A Subtlety about Joins

<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>

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

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

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

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

Which address?

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

```
SELECT  DISTINCT  x.pnome, y.address
FROM     Person AS x, Company AS y
WHERE    x.worksfor = y.cname
```
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
```
An Unintuitive Query

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

What does it compute?

Computes R ∩ (S ∪ T)  But what if S = φ?
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'
```

Beware of duplicates!
Removing Duplicates

```sql
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');
```

Now they are equivalent

```sql
SELECT DISTINCT Company.city
FROM Company, Product, Purchase
WHERE Company.name = Product.maker
    AND Product.pname = Purchase.product
    AND Purchase.buyer = 'Joe Blow'
```
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    Purchase
                      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?
Question for Database Fans and their Friends

• Answer: all SFW queries are monotone (figure out what this means). A query with ALL 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:

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

same as Count(*)

We probably want:

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

Purchase(product, date, price, quantity)

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

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

What do they mean?
Simple Aggregations

Purchase

<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>

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…
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</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>
3. SELECT

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

<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>

<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 Quereis

```
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.

```
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 C1
GROUP BY a_1, ..., a_k
HAVING C2

S = may contain attributes a_1, ..., a_k and/or any aggregates but NO OTHER ATTRIBUTES
C1 = is any condition on the attributes in R_1, ..., R_n
C2 = is any condition on aggregate expressions
General form of Grouping and Aggregation

```
SELECT  S
FROM    R_{1},...,R_{n}
WHERE   C1
GROUP BY a_{1},...,a_{k}
HAVING  C2
```

Evaluation steps:
1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes $a_{1},...,a_{k}$
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
1. INTERSECT and EXCEPT:

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

\[
\begin{align*}
\text{(SELECT R.A, R.B FROM R) \quad \text{INTERSECT} \quad \text{(SELECT S.A, S.B FROM S)}}
\end{align*}
\]

\[
\begin{align*}
\text{SELECT R.A, R.B FROM R WHERE EXISTS(\text{SELECT * FROM S WHERE R.A=S.A and R.B=S.B})}
\end{align*}
\]

\[
\begin{align*}
\text{(SELECT R.A, R.B FROM R) \quad \text{EXCEPT} \quad \text{(SELECT S.A, S.B FROM S)}}
\end{align*}
\]

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

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

Find all companies that make some products with price < 100

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 ≥ 100

```sql
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.cname IN (SELECT Product.company
FROM Product
WHERE Product.price >= 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 >= 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
```

This is SQL by a novice.
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
```

This is SQL by an expert

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
```
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)

Why both?
Two Examples

Store\( (\text{sid}, \text{sname}) \)
Product\( (\text{pid}, \text{pname}, \text{price}, \text{sid}) \)

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)
```

But may return multiple product names per store
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 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 \( x = \text{NULL} \) 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:

  false \( = \) 0
  unknown \( = \) 0.5
  true \( = \) 1
Null Values

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

**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>

### Additional 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(*)
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:

\[
\text{INSERT INTO } R(A_1, \ldots, A_n) \text{ VALUES } (v_1, \ldots, v_n)
\]

Example: Insert a new purchase to the database:

\[
\text{INSERT INTO } \text{Purchase}(\text{buyer, seller, product, store})
\text{VALUES } (\text{‘Joe’, ‘Fred’, ‘wakeup-clock-espresso-machine’, ‘The Sharper Image’})
\]

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

\[
\text{INSERT INTO PRODUCT(name)}
\]

\[
\text{SELECT DISTINCT Purchase.product}
\]

\[
\text{FROM Purchase}
\]

\[
\text{WHERE Purchase.date > "10/26/01"}
\]

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

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

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

Suppose database got corrupted and we need to fix it:

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>listPrice</td>
</tr>
<tr>
<td>gizmo</td>
<td>100</td>
</tr>
<tr>
<td>gizmo</td>
<td></td>
</tr>
<tr>
<td>camera</td>
<td></td>
</tr>
</tbody>
</table>

Task: insert in Product all prodNames from Purchase
Insertion: an Example

```
INSERT INTO Product(name)

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

<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

\[
\begin{align*}
\text{INSERT INTO} & \quad \text{Product(name, listPrice)} \\
\text{SELECT DISTINCT} & \quad \text{prodName, price} \\
\text{FROM} & \quad \text{Purchase} \\
\text{WHERE} & \quad \text{prodName NOT IN (SELECT name FROM Product)}
\end{align*}
\]

<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:

```sql
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:

```sql
UPDATE PRODUCT
SET price = price/2
WHERE Product.name IN
    (SELECT product
     FROM Purchase
     WHERE Date = 'Oct, 25, 1999');
```