Lectures 2&3: Introduction to SQL
Lecture 2: SQL Part I
Today’s Lecture

1. SQL introduction & schema definitions
   • ACTIVITY: Table creation

2. Basic single-table queries
   • ACTIVITY: Single-table queries!

3. Multi-table queries
   • ACTIVITY: Multi-table queries!
1. SQL Introduction & Definitions
What you will learn about in this section

1. What is SQL?
2. Basic schema definitions
3. Keys & constraints intro
4. ACTIVITY: CREATE TABLE statements
SQL Motivation

• Dark times 5 years ago.
  • Are databases dead?

• Now, as before: everyone sells SQL
  • Pig, Hive, Impala

• “Not-Yet-SQL?”
Basic SQL
SQL Introduction

• SQL is a standard language for querying and manipulating data

• SQL is a **very high-level** programming language
  • This works because it is optimized well!

• Many standards out there:
  • ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ....
  • Vendors support various subsets

*NB:* Probably the world’s most successful parallel programming language (multicore?)
SQL is a...

• Data Definition Language (DDL)
  • Define relational *schemata*
  • Create/alter/delete tables and their attributes

• Data Manipulation Language (DML)
  • Insert/delete/modify tuples in tables
  • Query one or more tables – discussed next!
Tables in SQL

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

A relation or table is a multiset of tuples having the attributes specified by the schema.

Let’s break this definition down.
Tables in SQL

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

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed)

List: [1, 1, 2, 3]
Set: {1, 2, 3}
Multiset: {1, 1, 2, 3}

i.e. no `next()`, etc. methods!
# Tables in SQL

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

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation.

**NB:** Attributes must have an **atomic type** in standard SQL, i.e. not a list, set, etc.
Tables in SQL

A **tuple** or **row** is a single entry in the table having the attributes specified by the schema.

---

### Product

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

*Also referred to sometimes as a record*
Tables in SQL

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

The number of tuples is the **cardinality** of the relation.

The number of attributes is the **arity** of the relation.
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?
Table Schemas

• The **schema** of a table is the table name, its attributes, and their types:

```
Product(Pname: string, Price: float, Category: string, Manufacturer: string)
```

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

```
Product(Pname: string, Price: float, Category: string, Manufacturer: string)
```
Key constraints

A **key** is a **minimal subset of attributes** that acts as a unique identifier for tuples in a relation

- A key is an implicit constraint on which tuples can be in the relation
  - i.e. if two tuples agree on the values of the key, then they must be the same tuple!

```
Students(sid:string, name:string, gpa: float)
```

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?
NULL and NOT NULL

• To say “don’t know the value” we use **NULL**
  • NULL has (sometimes painful) semantics, more detail later

Students(sid:string, name:string, gpa: float)

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Bob</td>
<td>3.9</td>
</tr>
<tr>
<td>143</td>
<td>Jim</td>
<td>NULL</td>
</tr>
</tbody>
</table>

*Say, Jim just enrolled in his first class.*

In SQL, we may constrain a column to be NOT NULL, e.g., “name” in this table
General Constraints

• We can actually specify arbitrary assertions
  • E.g. “There cannot be 25 people in the DB class”

• In practice, we don’t specify many such constraints. Why?
  • Performance!

Whenever we do something ugly (or avoid doing something convenient) it’s for the sake of performance
Summary of Schema Information

• Schema and Constraints are how databases understand the semantics (meaning) of data

• They are also useful for optimization

• SQL supports general constraints:
  • Keys and foreign keys are most important
  • We’ll give you a chance to write the others
ACTIVITY: Activity-2-1.ipynb
2. Single-table queries
What you will learn about in this section

1. The SFW query

2. Other useful operators: LIKE, DISTINCT, ORDER BY

3. ACTIVITY: Single-table queries
SQL Query

- Basic form (there are many many more bells and whistles)

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

Call this a SFW query.
Simple SQL Query: Selection

**Selection** is the operation of filtering a relation’s tuples on some condition.

### Example SQL Query

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

### 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>
Simple SQL Query: Projection

**Projection** is the operation of producing an output table with tuples that have a subset of their prior attributes.

```
SELECT Pname, Price, Manufacturer
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>

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>GizmoWorks</td>
</tr>
</tbody>
</table>
Notation

**Input schema**

`Product(PName, Price, Category, Manufacturer)`

**SELECT** Pname, Price, Manufacturer  
**FROM** Product  
**WHERE** Category = 'Gadgets'

**Output schema**

`Answer(PName, Price, Manufacturer)`
A Few Details

• **SQL commands** are case insensitive:
  - Same: SELECT, Select, select
  - Same: Product, product

• **Values** are not:
  - Different: ‘Seattle’, ‘seattle’

• Use single quotes for constants:
  - ‘abc’ - yes
  - “abc” - no
LIKE: Simple String Pattern Matching

\[
\begin{align*}
\text{SELECT} & \quad \ast \\
\text{FROM} & \quad \text{Products} \\
\text{WHERE} & \quad \text{PName} \ \text{LIKE} \ ‘%gizmo%’
\end{align*}
\]

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

SELECT DISTINCT Category
FROM Product

Versus

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>

Lecture 2 > Section 2 > Other operators
ORDER BY: Sorting 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.
ACTIVITY: Activity-2-2.ipynb
3. Multi-table queries
What you will learn about in this section

1. Foreign key constraints
2. Joins: basics
3. Joins: SQL semantics
4. ACTIVITY: Multi-table queries
Foreign Key constraints

• Suppose we have the following schema:

\[
\begin{align*}
\text{Students}(\text{sid}: \text{string}, \text{name}: \text{string}, \text{gpa}: \text{float}) \\
\text{Enrolled}(\text{student\_id}: \text{string}, \text{cid}: \text{string}, \text{grade}: \text{string})
\end{align*}
\]

• And we want to impose the following constraint:

- ‘Only bona fide students may enroll in courses’ i.e. a student must appear in the Students table to enroll in a class

<table>
<thead>
<tr>
<th>Students</th>
<th>Enrolled</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
</tr>
<tr>
<td>101</td>
<td>Bob</td>
</tr>
<tr>
<td>123</td>
<td>Mary</td>
</tr>
</tbody>
</table>

We say that student\_id is a foreign key that refers to Students
Declaring Foreign Keys

Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)

CREATE TABLE Enrolled(
    student_id CHAR(20),
    cid CHAR(20),
    grade CHAR(10),
    PRIMARY KEY (student_id, cid),
    FOREIGN KEY (student_id) REFERENCES Students
)
Foreign Keys and update operations

Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)

• What if we insert a tuple into Enrolled, but no corresponding student?
  • INSERT is rejected (foreign keys are constraints)!

• What if we delete a student?
  1. Disallow the delete
  2. Remove all of the courses for that student
  3. SQL allows a third via NULL (not yet covered)
# 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>

What is a foreign key vs. a key here?

## 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`

Ex: 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
```
Joins

Product(PName, Price, Category, Manufacturer)
Company(CName, StockPrice, Country)

Ex: 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

A join between tables returns all unique combinations of their tuples which meet some specified join condition.
Joins

Several equivalent ways to write a basic join in SQL:

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

```
SELECT PName, Price
FROM Product
JOIN Company ON Manufacturer = Cname
    AND Country='Japan'
WHERE Price <= 200
```

A few more later on...
Joins

### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manuf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29</td>
<td>Gadgets</td>
<td>GWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

### Company

<table>
<thead>
<tr>
<th>Cname</th>
<th>Stock</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GWorks</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
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer = CName
AND Country='Japan'
AND Price <= 200
```
Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)
Company(name, address)

SELECT DISTINCT name, address
FROM Person, Company
WHERE worksfor = name

Which “address” does this refer to?
Which “name”s??
Tuple Variable Ambiguity in Multi-Table

Person(name, address, worksfor)
Company(name, address)

SELECT DISTINCT Person.name, Person.address
FROM Person, Company
WHERE Person.worksfor = Company.name

Both equivalent ways to resolve variable ambiguity

SELECT DISTINCT p.name, p.address
FROM Person p, Company c
WHERE p.worksfor = c.name
Meaning (Semantics) of SQL Queries

```
SELECT x_1.a_1, x_1.a_2, ..., x_n.a_k
FROM R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE Conditions(x_1,..., x_n)
```

Answer = {}
for x_1 in R_1 do
    for x_2 in R_2 do
        ..... 
        for x_n in R_n do
            if Conditions(x_1,..., x_n) 
                then Answer = Answer ∪ {(x_1.a_1, x_1.a_2, ..., x_n.a_k)}
return Answer

Almost never the fastest way to compute it!

Note: this is a multiset union
An example of SQL semantics

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Cross Product

Apply Selections / Conditions

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>

Apply Projection

Output

<table>
<thead>
<tr>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>
Note the **semantics** of a join

1. **Take cross product:**
   
   $$X = R \times S$$

2. **Apply selections / conditions:**
   
   $$Y = \{(r, s) \in X \mid r.A == r.B\}$$

3. **Apply projections** to get final output:
   
   $$Z = (y.A,) \text{ for } y \in Y$$

Recall: Cross product \((A \times B)\) is the set of all unique tuples in \(A, B\)

Ex: \(\{a, b, c\} \times \{1, 2\}\)

\[ = \{(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)\} \]

= Filtering!

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on...)

```sql
SELECT R.A
FROM R, S
WHERE R.A = S.B
```
Note: we say “semantics” not “execution order”

• The preceding slides show *what a join means*

• Not actually how the DBMS executes it under the covers
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

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PName</strong></td>
<td><strong>Cname</strong></td>
</tr>
<tr>
<td><strong>Price</strong></td>
<td><strong>Stock</strong></td>
</tr>
<tr>
<td><strong>Category</strong></td>
<td><strong>Country</strong></td>
</tr>
<tr>
<td><strong>Manuf</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Gizmo</th>
<th>GWorks</th>
<th>Gadgets</th>
<th>25</th>
<th>USA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Powergizmo</td>
<td>GWorks</td>
<td>Gadgets</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>Canon</td>
<td>Photography</td>
<td>15</td>
<td>Japan</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>Hitachi</td>
<td>Household</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>

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

What is the problem?
What’s the solution?
ACTIVITY: Lecture-2-3.ipynb
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?
An Unintuitive Query

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

Computes \( R \cap (S \cup T) \)

But what if \( S = \phi \)?

Go back to the semantics!
An Unintuitive Query

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

• Recall the semantics!
  1. Take cross-product
  2. Apply selections / conditions
  3. Apply projection

• If S = {}, then the cross product of R, S, T = {}, and the query result = {}!

Must consider semantics here. Are there more explicit way to do set operations like this?
The coolest people

JOIN TOGETHER

multiple tables with advanced SQL!!!

Lecture 3: SQL Part II
Course announcements

• PS1 is posted online and due at beginning of class on 10/6 - start early!

• We will now be using an online queue management system for OHs - see details on Piazza
  - http://queuestatus.com/organizations/1/queues/6

• We will write the lecture attendance code on the whiteboard sometime during lecture

Created by a former CS145 student + CA using databases!!!
A note on Piazza posting

• Please post questions **publicly** if possible

  • We will prioritize answering public questions, and **reserve the right to make private posts public**

  • You can always post anonymously if more comfortable!

  • We want everyone to be on the same page, and benefit from others’ questions
Today’s Lecture

1. Set operators & nested queries
   • ACTIVITY: Set operator subtleties

2. Aggregation & GROUP BY
   • ACTIVITY: Fancy SQL Part I

3. Advanced SQL-izing
   • ACTIVITY: Fancy SQL Part II
1. Set Operators & Nested Queries
What you will learn about in this section

1. Multiset operators in SQL
2. Nested queries
3. ACTIVITY: Set operator subtleties
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?
An Unintuitive Query

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

Computes $R \cap (S \cup T)$

But what if $S = \phi$?

Go back to the semantics!
An Unintuitive Query

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

• Recall the semantics!
  1. Take cross-product
  2. Apply selections / conditions
  3. Apply projection

• If S = {}, then the cross product of R, S, T = {}, and the query result = {}!

Must consider semantics here.
Are there more explicit way to do set operations like this?
What does this look like in Python?

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

- **Semantics:**
  1. Take cross-product
  2. Apply selections / conditions
  3. Apply projection

*Joins / cross-products are just nested for loops (in simplest implementation)!*

*If-then statements!*
What does this look like in Python?

```python
output = {}
for r in R:
    for s in S:
        for t in T:
            if r['A'] == s['A'] or r['A'] == t['A']:
                output.add(r['A'])
return list(output)
```

Can you see now what happens if S = []?

See bonus activity on website!
Multiset Operations
Recall Multisets

Multiset $X$

<table>
<thead>
<tr>
<th>Tuple</th>
<th>Equivalent Representations of a Multiset</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(1, a)$</td>
<td></td>
</tr>
<tr>
<td>$(1, a)$</td>
<td></td>
</tr>
<tr>
<td>$(1, b)$</td>
<td></td>
</tr>
<tr>
<td>$(2, c)$</td>
<td></td>
</tr>
<tr>
<td>$(2, c)$</td>
<td></td>
</tr>
<tr>
<td>$(2, c)$</td>
<td></td>
</tr>
<tr>
<td>$(1, d)$</td>
<td></td>
</tr>
<tr>
<td>$(1, d)$</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** In a set all counts are $\{0,1\}$.

\[ \lambda(X) = \text{“Count of tuple in } X\text{”} \]

(Items not listed have implicit count 0)
Generalizing Set Operations to Multiset Operations

For sets, this is intersection.
Generalizing Set Operations to Multiset Operations

Multiset $X$

<table>
<thead>
<tr>
<th>Tuple</th>
<th>$\lambda(X)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>2</td>
</tr>
<tr>
<td>(1, b)</td>
<td>0</td>
</tr>
<tr>
<td>(2, c)</td>
<td>3</td>
</tr>
<tr>
<td>(1, d)</td>
<td>0</td>
</tr>
</tbody>
</table>

Multiset $Y$

<table>
<thead>
<tr>
<th>Tuple</th>
<th>$\lambda(Y)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, a)</td>
<td>5</td>
</tr>
<tr>
<td>(1, b)</td>
<td>1</td>
</tr>
<tr>
<td>(2, c)</td>
<td>2</td>
</tr>
<tr>
<td>(1, d)</td>
<td>2</td>
</tr>
</tbody>
</table>

$\bigcup = \lambda(Z) = \lambda(X) + \lambda(Y)$

For sets, this is **union**.
Multiset Operations in SQL
Explicit Set Operators: INTERSECT

\[
\text{SELECT } R.A \\
\text{FROM } R, S \\
\text{WHERE } R.A=S.A \\
\text{INTERSECT} \\
\text{SELECT } R.A \\
\text{FROM } R, T \\
\text{WHERE } R.A=T.A
\]
UNION

\[
\begin{align*}
\text{SELECT} & \quad \text{R.A} \\
\text{FROM} & \quad \text{R, S} \\
\text{WHERE} & \quad \text{R.A} = \text{S.A} \\
\text{UNION} & \\
\text{SELECT} & \quad \text{R.A} \\
\text{FROM} & \quad \text{R, T} \\
\text{WHERE} & \quad \text{R.A} = \text{T.A}
\end{align*}
\]

\[\{r.A | r.A = s.A\} \cup \{r.A | r.A = t.A\}\]

Why aren’t there duplicates?

What if we want duplicates?
UNION ALL

\[
\begin{align*}
\text{SELECT} & \quad R.A \\
\text{FROM} & \quad R, S \\
\text{WHERE} & \quad R.A = S.A \\
\text{UNION ALL} & \quad \\
\text{SELECT} & \quad R.A \\
\text{FROM} & \quad R, T \\
\text{WHERE} & \quad R.A = T.A
\end{align*}
\]

\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}

ALL indicates Multiset operations
EXCEPT

\[
\begin{align*}
\text{SELECT} & \quad R.A \\
\text{FROM} & \quad R, S \\
\text{WHERE} & \quad R.A=S.A \\
\text{EXCEPT} & \quad \{r.A | r.A = s.A\}\{r.A | r.A = t.A\}
\end{align*}
\]

What is the multiset version?
INTERSECT: Still some subtle problems...

Company(name, hq_city)
Product(pname, maker, factory_loc)

```
SELECT hq_city
FROM Company, Product
WHERE maker = name
AND factory_loc = 'US'
INTERSECT
SELECT hq_city
FROM Company, Product
WHERE maker = name
AND factory_loc = 'China'
```

“What headquarters of companies which make gizmos in US AND China”

What if two companies have HQ in US: BUT one has factory in China (but not US) and vice versa? **What goes wrong?**
INTERSECT: Remember the semantics!

Example: \( C \times X \times P = \)

<table>
<thead>
<tr>
<th>C.name</th>
<th>C.hq_city</th>
<th>P.pname</th>
<th>P.maker</th>
<th>P.factory_loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Co.</td>
<td>U.S.</td>
<td>X</td>
<td>X Co.</td>
<td>U.S.</td>
</tr>
<tr>
<td>X Co.</td>
<td>U.S.</td>
<td>Y</td>
<td>Y Inc.</td>
<td>China</td>
</tr>
<tr>
<td>Y Inc.</td>
<td>U.S.</td>
<td>Y</td>
<td>Y Inc.</td>
<td>China</td>
</tr>
<tr>
<td>Y Inc.</td>
<td>U.S.</td>
<td>X</td>
<td>X Co.</td>
<td>U.S.</td>
</tr>
</tbody>
</table>
**INTERSECT: Remember the semantics!**

Company(name, hq_city) AS C
Product(pname, maker, factory_loc) AS P

```sql
SELECT hq_city
FROM Company, Product
WHERE maker = name
  AND factory_loc = 'US'
INTERSECT
SELECT hq_city
FROM Company, Product
WHERE maker = name
  AND factory_loc = 'China'
```

Example: \( C \times X \times P = \)

<table>
<thead>
<tr>
<th>C.name</th>
<th>C.hq_city</th>
<th>P.pname</th>
<th>P.maker</th>
<th>P.factory_loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Co.</td>
<td>U.S.</td>
<td>X</td>
<td>X Co.</td>
<td>U.S.</td>
</tr>
<tr>
<td>X Co.</td>
<td>U.S.</td>
<td>Y</td>
<td>Y Inc.</td>
<td>China</td>
</tr>
<tr>
<td>Y Inc.</td>
<td>U.S.</td>
<td>Y</td>
<td>Y Inc.</td>
<td>China</td>
</tr>
<tr>
<td>Y Inc.</td>
<td>U.S.</td>
<td>X</td>
<td>X Co.</td>
<td>U.S.</td>
</tr>
</tbody>
</table>

Hq_city = U.S.  
Hq_city = U.S.
INTERSECT: Remember the semantics!

**Example:**

```
C * X * P =
```

<table>
<thead>
<tr>
<th>C.name</th>
<th>C.hq_city</th>
<th>P.pname</th>
<th>P.maker</th>
<th>P.factory_loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>X Co.</td>
<td>U.S.</td>
<td>X</td>
<td>X Co.</td>
<td>U.S.</td>
</tr>
<tr>
<td>X Co.</td>
<td>U.S.</td>
<td>Y</td>
<td>Y Inc.</td>
<td>China</td>
</tr>
<tr>
<td>Y Inc.</td>
<td>U.S.</td>
<td>Y</td>
<td>Y Inc.</td>
<td>China</td>
</tr>
<tr>
<td>Y Inc.</td>
<td>U.S.</td>
<td>X</td>
<td>X Co.</td>
<td>U.S.</td>
</tr>
</tbody>
</table>

We did the INTERSECT on the wrong attributes!
One Solution: **Nested Queries**

Company\((name, hq\_city)\)  
Product\((pname, maker, factory\_loc)\)

```sql
SELECT DISTINCT hq\_city  
FROM Company, Product  
WHERE maker = name  
AND name IN (  
    SELECT maker  
    FROM Product  
    WHERE factory\_loc = 'US')  
AND name IN (  
    SELECT maker  
    FROM Product  
    WHERE factory\_loc = 'China')
```

“Headquarters of companies which make gizmos in US AND China”

Note: If we hadn’t used DISTINCT here, how many copies of each hq\_city would have been returned?
High-level note on nested queries

• We can do nested queries because SQL is *compositional*:
  
  • Everything (inputs / outputs) is represented as multisets - the output of one query can thus be used as the input to another (nesting)!

• This is extremely powerful!
Nested queries: Sub-queries Returning Relations

Another example:

```
SELECT c.city
FROM Company c
WHERE c.name IN (SELECT pr.maker
                   FROM Purchase p, Product pr
                   WHERE p.product = pr.name
                    AND p.buyer = 'Joe Blow')
```

“Cities where one can find companies that manufacture products bought by Joe Blow”
Nested Queries

Is this query equivalent?

```
SELECT c.city
FROM Company c,
     Product pr,
     Purchase p
WHERE c.name = pr.maker
  AND pr.name = p.product
  AND p.buyer = 'Joe Blow'
```

Beware of duplicates!
Nested Queries

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

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

Now they are equivalent
Subqueries Returning Relations

You can also use operations of the form:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- $\exists R$

Ex: $\text{Product(name, price, category, maker)}$

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')
```
Subqueries Returning Relations

You can also use operations of the form:

- \( s > \text{ALL} \ R \)
- \( s < \text{ANY} \ R \)
- \( \text{EXISTS} \ R \)

Ex: \( \text{Product(name, price, category, maker)} \)

```
SELECT p1.name
FROM Product p1
WHERE p1.maker = 'Gizmo-Works'
AND EXISTS(
    SELECT p2.name
    FROM Product p2
    WHERE p2.maker <> 'Gizmo-Works'
    AND p1.name = p2.name)
```

Find ‘copycat’ products, i.e. products made by competitors with the same names as products made by “Gizmo-Works”
Nested queries as alternatives to INTERSECT and EXCEPT

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

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

INTERSECT and EXCEPT not in some DBMSs!

If R, S have no duplicates, then can write without sub-queries (HOW?)
A question for Database Fans & Friends

• Can we express the previous nested queries as single SFW queries?

• Hint: show that all SFW queries are monotone (roughly: more tuples, more answers).
  • A query with ALL is not monotone
Correlated Queries

```
Movie(title, year, director, length)

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

Find movies whose title appears more than once.

Note the scoping of the variables!

Note also: this can still be expressed as single SFW query...
Complex Correlated Query

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

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

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

Can be very powerful (also much harder to optimize)
Basic SQL Summary

• SQL provides a high-level declarative language for manipulating data (DML)

• The workhorse is the SFW block

• Set operators are powerful but have some subtleties

• Powerful, nested queries also allowed.
Activity-3-1.ipynb
2. Aggregation & GROUP BY
What you will learn about in this section

1. Aggregation operators
2. GROUP BY
3. GROUP BY: with HAVING, semantics
4. ACTIVITY: Fancy SQL Pt. 1
Aggregation

\[
\text{SELECT} \ \text{AVG}(\text{price}) \\
\text{FROM} \ \text{Product} \\
\text{WHERE} \ \text{maker} = \text{"Toyota"}
\]

\[
\text{SELECT} \ \text{COUNT}(*) \\
\text{FROM} \ \text{Product} \\
\text{WHERE} \ \text{year} > 1995
\]

- 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

```sql
SELECT COUNT(category) 
FROM Product 
WHERE year > 1995
```

Note: Same as COUNT(*).

Why?

We probably want:

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

Purchase(product, date, price, quantity)

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

What do these mean?

```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```
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 (= 1*20 + 1.50*20)
Grouping and Aggregation

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

Semantics of the query:

1. Compute the **FROM** and **WHERE** clauses

2. Group by the attributes in the **GROUP BY**

3. Compute the **SELECT** clause: grouped attributes and aggregates
1. Compute the **FROM** and **WHERE** clauses

```sql
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>
### 2. Group by the attributes in the **GROUP BY**

The SQL query to group by the product and calculate the total sales is:

```sql
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>

The grouped data shows:

<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. Compute the **SELECT** clause: grouped attributes and aggregates

```sql
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></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></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'
```
HAVING Clause

```
SELECT product, SUM(price*quantity)  
FROM Purchase  
WHERE date > '10/1/2005'  
GROUP BY product  
HAVING SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers.

**HAVING** clauses contain conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples...***
General form of Grouping and Aggregation

SELECT S
FROM R₁,…,Rₙ
WHERE C₁
GROUP BY a₁,…,aₖ
HAVING C₂

- S = Can ONLY contain attributes a₁,…,aₖ and/or aggregates over other attributes
- C₁ = is any condition on the attributes in R₁,…,Rₙ
- C₂ = is any condition on the aggregate expressions

Why?
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
```

Evaluation steps:
1. Evaluate **FROM-WHERE**: apply condition $C_1$ on the attributes in $R_1,...,R_n$
2. **GROUP BY** the attributes $a_1,...,a_k$
3. Apply condition $C_2$ to each group (may have aggregates)
4. Compute aggregates in $S$ and return the result
Group-by v.s. Nested Query

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

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

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

• 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
```

No need for DISTINCT: automatically from GROUP BY
Group-by vs. Nested Query

Which way is more efficient?

- Attempt #1- *With nested*: How many times do we do a SFW query over all of the Wrote relations?
  
- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be much more efficient!
Activity-3-2.ipynb
3. Advanced SQL-izing
What you will learn about in this section

1. Quantifiers
2. NULLs
3. Outer Joins
4. ACTIVITY: Fancy SQL Pt. II
Quantifiers

Product(name, price, company)
Company(name, city)

Find all companies that make some products with price < 100

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

An existential quantifier is a logical quantifier (roughly) of the form “there exists”

Existential: easy 😊
Quantifiers

Product(name, price, company)
Company(name, city)

Find all companies with products all having price < 100

SELECT DISTINCT Company.cname
FROM Company
WHERE Company.name NOT IN(
    SELECT Product.company
    FROM Product.price >= 100)

A universal quantifier is of the form “for all”

Find all companies that make only products with price < 100

Equivalent

Universal: hard ! 😞
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

• *For numerical operations*, NULL -> NULL:
  - If \( x = \text{NULL} \) then \( 4*(3-x)/7 \) is still NULL

• *For boolean operations*, in SQL there are three values:

  \[
  \begin{align*}
  \text{FALSE} &= 0 \\
  \text{UNKNOWN} &= 0.5 \\
  \text{TRUE} &= 1
  \end{align*}
  \]

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

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

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

Won’t return e.g. (age=20 height=NULL weight=200)!

Rule in SQL: include only tuples that yield TRUE / 1.0
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!
RECAP: Inner Joins

By default, joins in SQL are "inner joins":

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

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

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

Both equivalent: Both INNER JOINS!
Inner Joins + NULLS = Lost data?

By default, joins in SQL are "inner joins":

\[
\text{Product(name, category)}
\]
\[
\text{Purchase(prodName, store)}
\]

\[
\text{SELECT Product.name, Purchase.store}
\]
\[
\text{FROM Product}
\]
\[
\text{JOIN Purchase ON Product.name = Purchase.prodName}
\]

\[
\text{SELECT Product.name, Purchase.store}
\]
\[
\text{FROM Product, Purchase}
\]
\[
\text{WHERE Product.name = Purchase.prodName}
\]

However: Products that never sold (with no Purchase tuple) will be lost!
Outer Joins

• An outer join returns tuples from the joined relations that don’t have a corresponding tuple in the other relations
  • I.e. If we join relations A and B on \( a.X = b.X \), and there is an entry in A with \( X=5 \), but none in B with \( X=5 \)...  
    • A LEFT OUTER JOIN will return a tuple (a, NULL)!

• Left outer joins in SQL:

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

Now we’ll get products even if they didn’t sell
INNER JOIN:

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

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

Note: another equivalent way to write an INNER JOIN!
LEFT OUTER JOIN:

**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>

**SQL Query**

```
SELECT Product.name, Purchase.store
FROM Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
```
Other 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
Activity-3-3.ipynb
Summary

SQL is a rich programming language that handles the way data is processed *declaratively*.