Today…

- Quick recap of basic SQL
- Advanced querying features
  - Set membership, existence, nested queries
  - Inner joins and Outer joins in MySQL
- Aggregate functions on sets of tuples
  - Max, sum,….
- Operating on partitions of sets/relations
  - GROUPBY

Next week….

- How to design a good schema?
- 3-tier application design: Web + PHP + MySQL

Basic SQL and Relational Algebra

- The SELECT statement can be mapped directly to relational algebra.

  - SELECT $A_1, A_2, \ldots, A_n$ /* this is projection */
  - FROM $R_1, R_2, \ldots, R_m$ /* this is the selection op */
  - WHERE $P$ /* this is cartesian product */

  is equivalent to:

  $$\Pi_{A_1, A_2, \ldots, A_n}(\sigma_P(R_1 \times R_2 \times \ldots \times R_m))$$

SQL….review basic operations

- Select clause
- Need to specify Join condition
- Concept of ‘aliasing’ to rename relation using AS keyword
  - Rename an attribute…in Select clause
  - Rename a relation
- Important: can leave out the AS keyword!

- INTO clause
  - Variation on aliasing
  - Pipe output of SELECT into another table
Renaming and Aliasing

- Renaming is also used when two or more copies of the same table are in a query. Using aliases allows you to uniquely identify which table you are talking about.

Example: Return the employees and their managers where the managers make less than the employee.

```sql
SELECT E.ename, M.ename
FROM emp as E, emp as M
WHERE E.supereno = M.eno and E.salary > M.salary
```

- There is another (more elegant) interpretation that accomplishes the same using concept of "tuple" variables
  - Taken from relational calculus

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 it can be null (nullable attribute) or not
- NOT NULL after declaring attribute domain
- 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:

<table>
<thead>
<tr>
<th>Boolean Value</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>0.5</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
</tr>
</tbody>
</table>

**Rule in SQL:** include only tuples that yield TRUE

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

E.g.

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

Rule in SQL: include only tuples that yield TRUE
Null Values

Unexpected behavior:

```sql
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`

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

Now it includes all Persons

IN Operator

- To specify that an attribute value should be in a given set of values, the **IN** keyword is used.
- Example: Return all employees who are in any one of the departments {'D1', 'D2', 'D3'}.
  ```sql
  SELECT ename 
  FROM emp 
  WHERE dno IN ('D1', 'D2', 'D3') 
  ```
- Note that this is equivalent to using OR:
  ```sql
  SELECT ename 
  FROM emp 
  WHERE dno = 'D1' OR dno = 'D2' OR dno = 'D3' 
  ```
- more practical uses of **IN** and **NOT IN** when we study nested subqueries.

Set Operations

- The set operations of union, intersection, and difference are used to combine the results of two SQL queries.
- **UNION, INTERSECT, EXCEPT**
- Note: **UNION ALL** returns all rows
- Example: Return the sid of students who are either taking course with cid=123 or course with cid=345.
  ```sql
  (SELECT sid 
   FROM students 
   WHERE cid='123') 
  UNION 
  (SELECT sid 
   FROM students 
   WHERE cid = '345'); 
  ```
Set Operations

- MySQL does NOT support MINUS (EXCEPT) and INTERSECT
  - Can implement using other operators

Exercise 1

- Write these solutions…do not code.
  Using set operations, union and IN, provide MySQL queries for:
  - Ques 1: Find customers who have a loan or a deposit at Foggy Bottom branch.
  - Ques 2: Find names and IDs of customers who have a loan or deposit at the Foggy Bottom branch.

More SQL stuff …

- Nested Queries
- Set membership

Aliasing/Renaming and Tuple Variables

- Aliasing is nothing but use of tuple variables borrowed from relational calculus
- Concept of tuple/range variables in rel. calculus
  - Tuple \( t \) of type \( R \):
    - \( t \in R \)
    - \( \text{What about } x \in R, y \in R \)
    - One variable with name \( x \) and one with name \( y \)
    - Both have the same ‘type’

- Need to worry about scope of tuple variables when we have nested queries
Meaning (Semantics) of SQL Queries with tuples

Sometimes this is easier to map to how you develop your solution logic - due to similarity with high level languages

$$\begin{align*}
\text{Answer} &= \emptyset \\
\text{for } x_1 \text{ in } R_1 & \text{ do} \\
& \quad \text{for } x_2 \text{ in } R_2 \text{ do} \\
& \quad \quad \text{...} \\
& \quad \quad \text{for } x_n \text{ in } R_n \text{ do} \\
& \quad \quad \quad \text{if Conditions} \\
& \quad \quad \quad \text{then } \text{Answer} = \text{Answer} \cup \{ (a_1,\ldots,a_k) \} \\
\text{return } \text{Answer}
\end{align*}$$

Tuple variables

- Find customers (ID) who have an account at a branch where CustID 6666 has an account.

SELECT A.CustID
FROM Deposit A, Deposit B
WHERE B.CustID=6666
   AND A.branchname=B.branchname
- B is a variable that refers to Customer 6666
- Variable A can take on any values that satisfy the query
  The type of A is schema of Deposit

Subqueries

- SQL allows a single query to have multiple subqueries nested inside of it. This allows for more complex queries to be written.

- When queries are nested, the outer statement determines the contents of the final result, while the inner SELECT statements are used by the outer statement (often to lookup values for WHERE clauses).

- Need to be careful about scope of tuple variables
  Scoping rules: local definition and then global
  In subquery – legal to use only tuple variables defined in subquery itself or in any query that contains the subquery

Nested Queries: Semantics and set operators

- Evaluate subquery at each reference
  - Construct cross product of tables in FROM clause
  - For each row when testing predicate conditions in WHERE clause
    Recompute subquery
    – Is this really necessary?

- Set membership operators provided to test results of subquery
  - IN, EXISTS, CONTAINS (subset), op ALL, op SOME …(op is >, <, =)
Subqueries Returning Relations and Set Membership Operations

- Can check for set membership using IN and NOT IN
- NOT IN limits for absence of tuples
- IN membership tests for membership in the set A
  - Set A must be produced by a SELECT
- NOTIN tests for absence of tuples
  - Can test using multiple attribute elements
- Set existence using EXISTS
  - Returns true if the argument subquery is non-empty (the converse for NOT EXISTS)
  - Checking for empty relations

Set Membership: Quantifiers

- Find all companies that make some products with price < 100

Subqueries Returning Relations

- Write a SELECT query to obtain this set
- Return cities of companies that manufacture products bought by Joe Plumber

Company(name, city)
Product(name, price, company)
Purchase(id, product, buyer)

Return cities of companies that manufacture products bought by Joe Plumber

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');

Return cities of companies that manufacture products bought by Joe Plumber

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

Return cities of companies that manufacture products bought by Joe Plumber

SELECT Company.city
FROM Company
WHERE Company.name IN
(SELECT Product.maker
FROM Purchase, Product
WHERE Product.pname=Purchase.product
AND Purchase.buyer = 'Joe Plumber');
Set Membership: Quantifiers

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

Find all companies that make some products with price < 100

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

Existential: easy ! 😊

Set Membership: Quantifiers

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

Find all companies that make only products with price < 100

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

Universal: hard ! 😞

Set Membership: 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)
```

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

Set Membership: Quantifiers

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

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

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

2. Find all companies s.t. all their products have price < 100
More Set Membership Operations

- Previous operators allowed checking for existence
- SQL provides operators to test elements of one set A with elements on another set B
  - SOME: \( \text{op} \text{ SOME} \)
    - Also called as ANY in some versions
  - ALL: \( \text{op} \text{ ALL} \)
    - \( \text{op} \) can be \( \geq, >, <, <=, =, \text{not=} \)
- Test single value against members of an entire set
  - \( X > \text{ALL} (R) \)

Set Comparison Operations: subset

- Check if one set (query result) contains another set (query result)
  - Is A subset of B?
  - Is A not a subset of B?
- \( \text{contains} \) and \( \text{not contains} \) operators

Exercise 1..continued

Use the set membership and set existence operators for the queries below:

- Find all customers (IDs) who have both a loan and an account at the Foggy Bottom branch.
  - This is how you can do an intersection operation in MySQL
- Find all customers (IDs) who have an account but no loan at the bank.
  - Set difference operator
- Find branches that have greater assets than some branch located in DC
- Find all customers (IDs) who have an account at some branch located in DC

Even more SQL....

- Functions on sets of tuples
  - Aggregate functions: Max, sum,.....
- Operating on partitions of sets/relations
  - GROUPBY
### SQL— Aggregate Operations

- Thus far SQL (and Relational Algebra/Calculus) only fetched data stored in database tables.
- What if we need some basic ‘statistics’ on the data?
  - Number of rows?
  - Maximum value in a field?
- Aggregate Operators: apply a function to a set of tuples
  - Function defined on one (or more) field
  - Number of customers with loans
  - Average balance for a customer
  - Number of tuples in a relation
  - …

### Aggregate Operators

- Compute functions on set of tuples selected by where clause
  - Operate on a single column
- Semantics: if SELECT clause contains aggregate operations then it can contain only aggregate operations
  - Except when groupby construct is used
  - Functions on sets of values but result is single value
  - Average, minimum, maximum, sum, count(size)
- These functions operate on a single column of a table and return a single value.

### Aggregate Functions

- The five basic aggregate functions are:
  - COUNT - returns the # of values in a column
  - SUM - returns the sum of the values in a column
  - AVG - returns the average of the values in a column
  - MIN - returns the smallest value in a column
  - MAX - returns the largest value in a column

- Notes:
  1) COUNT, MAX, and MIN apply to all types of fields, whereas SUM and AVG apply to only numeric fields.
  2) Except for COUNT(*) all functions ignore nulls. COUNT(*) returns the number of rows in the table.
  3) Use DISTINCT to eliminate duplicates.

### Aggregate Function Example

- Return the number of employees and their average salary.

```sql
SELECT COUNT(eno) AS numEmp, AVG(salary) AS avgSalary
FROM emp
```

<table>
<thead>
<tr>
<th>numEmp</th>
<th>avgSalary</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>38750</td>
</tr>
</tbody>
</table>

Result
Exercise 2
- Find average balance across all accounts in the bank
- Find maximum loan given by the bank
- Find number of customers with accounts at GW branch
- Find the customer (ID) with the largest balance.

Aggregate Operators
- Significant extension of relational algebra.
  - COUNT (*)
  - COUNT ([DISTINCT] A)
  - SUM ([DISTINCT] A)
  - AVG ([DISTINCT] A)
  - MAX (A)
  - MIN (A)

Examples
Aggregate operators and computed columns
-arithmetic on column values

Purchase(product, date, price, quantity)

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)
Simple Aggregations

<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>

\[
\text{SELECT } \text{Sum(price } \times \text{ quantity)}
\text{ FROM Purchase}
\text{ WHERE product = 'bagel'} \Rightarrow 50 \ (= 20+30)
\]

Just a little bit more SQL.....

- Grouping
  - Operating on groups (partitions) of tuples

Motivation for Grouping

- So far, we've applied aggregate operators to all (qualifying) tuples. Sometimes, we want to apply them to each of several groups of tuples.
- Consider: Find the average balance for each branch in the bank.
  - In general, we don't know how many branches exist, and what the balances are!
  - Suppose we know that 10 branchnames exist; then we can write 10 queries that look like this (!):

  \[
  \text{For } x = 1, 2, \ldots, 10: \quad \text{SELECT } \text{AVG(balance)}
  \text{ FROM Deposit D}
  \text{ WHERE D.branchname=’x’}
  \]

Oops…no For loops in SQL!!

GROUP BY Clause

- Aggregate functions are most useful when combined with the GROUP BY clause. The GROUP BY clause groups the tuples based on the values of the attributes specified.
- When used in combination with aggregate functions, the result is a table where each tuple consists of unique values for the group by attributes and the result of the aggregate functions applied to the tuples of that group.
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&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>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>15</td>
</tr>
</tbody>
</table>

1. Compute the FROM and WHERE clauses.
2. Group by the attributes in the GROUPBY
3. Compute the SELECT clause: grouped attributes and aggregates.
GROUP BY Example

- For each employee title, return the number of employees with that title, and the minimum, maximum, and average salary.

```sql
SELECT title, COUNT(eno) AS numEmp,
       MIN(salary) as minSal,
       MAX(salary) as maxSal,
       AVG(salary) as avgSal
FROM emp
GROUP BY title
```

<table>
<thead>
<tr>
<th>title</th>
<th>numEmp</th>
<th>minSal</th>
<th>maxSal</th>
<th>avgSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>EE</td>
<td>2</td>
<td>30000</td>
<td>30000</td>
<td>30000</td>
</tr>
<tr>
<td>SA</td>
<td>3</td>
<td>50000</td>
<td>50000</td>
<td>50000</td>
</tr>
<tr>
<td>ME</td>
<td>2</td>
<td>40000</td>
<td>40000</td>
<td>40000</td>
</tr>
<tr>
<td>PR</td>
<td>1</td>
<td>20000</td>
<td>20000</td>
<td>20000</td>
</tr>
</tbody>
</table>

GROUP BY Clause Rules

- There are a few rules for using the GROUP BY clause:
  - 1) A column name cannot appear in the SELECT part of the query unless it is part of an aggregate function or in the list of group by attributes.
    - Note that the reverse is allowed: a column can be in the GROUP BY without being in the SELECT part.
  - 2) Any WHERE conditions are applied before the GROUP BY and aggregate functions are calculated.

Condition on the Groups

- What if we are only interested in groups that satisfy a condition?

HAVING Clause

- The HAVING clause is applied AFTER the GROUP BY clause and aggregate functions are calculated.
  - It is used to filter out entire groups that do not match certain criteria.
  - The HAVING clause can contain any condition that references aggregate functions and the group by attributes themselves.
  - However, any conditions on the GROUP BY attributes should be specified in the WHERE clause if possible due to performance reasons.
Grouping and Aggregation: Evaluation Steps

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…

HAVING Clause

Purchase(product, date, price, quantity)

Find total sales after 10/1/2005 per product, except that we consider only products that had at least 30 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.

Grouping and Aggregation

```
<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>
```

What if we are only interested in products that sold quantity >30?

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

Find total sales after 10/1/2005 per product, except that we consider only products that had at least 30 buyers.

HAVING clause contains conditions on aggregates.

General form of Grouping and Aggregation

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

Why ?

S = may contain attributes a1, ..., ak and/or any aggregate function but NO OTHER ATTRIBUTES

C1 = is any condition on the attributes in R1, ..., Rn

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

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

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

Generalized SELECT: Queries With GROUP BY and HAVING

- The attribute-list contains (i) attribute names (ii) terms with aggregate operations (e.g., MIN (balance)).
- The attribute list must be a subset of grouping-list. Intuitively, each answer tuple corresponds to a group, and these attributes must have a single value per group. (A group is a set of tuples that have the same value for all attributes in grouping-list.)

Conceptual Evaluation

- The cross-product of relation-list is computed, tuples that fail qualification in WHERE clause are discarded, `unnecessary' fields are deleted, and the remaining tuples are partitioned into groups by the value of attributes in grouping-list.
- The group-qualification specified in the HAVING clause is then applied to eliminate some groups. Expressions in HAVING clause must have a single value per group!
  - In effect, an attribute in group-qualification that is not an argument of an aggregate op also appears in grouping-list. (SQL does not exploit primary key semantics here!)
- One answer tuple is generated per qualifying group.
- Any aggregate function can be applied to a group
  - Final SELECT can have function over each selected group

A quick Note:
Group-by v.s. Nested Query

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

This is SQL by a novice
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
```

This is SQL by an expert

No need for DISTINCT: automatically from GROUP BY

Exercise 3

- Find average balance at each branch
- Find the number of customers at each branch
- Find branches, and their average balance, where the average balance is greater than 1200
- Find average balance, and branch names, of branches with at least three customers
- Find average balance of depositors (i.e., account holders) who live in Philadelphia and have at least two accounts.
- Find the branch(s) with the highest average balance

Uh, oh ! just a little bit more SQL..

- INNER JOIN
  - 'standard' join
- OUTER JOIN

Inner Join Operation

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
```
Why provide Inner Join?
- The semantics of the basic SQL query has cross product of the tables
  - Could be a very large intermediate result and impacts performance
  - Code optimizer (query processor) has to determine the join condition from the where clause
- Specifying join condition explicitly makes it easier for query optimizer to interpret
  - Creates the join instead of cross product
  - Smaller intermediate result, so better performance

Outerjoins
- Sometimes we may want to keep tuples that do not join with tuples with the other table
  - Find sales of all products, including those that with no sales
  - Find customers with or without accounts at the bank

The Inner Join result...
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
```

But Products that never sold will be lost!

Outerjoins
Sometimes we may want to keep tuples that do not join with the other table

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
```
### 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

### Quick Note: Outer Joins Application

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

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

```sql
SELECT Product.name, count(*)
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
  and Purchase.month = 'September'
GROUP BY Product.name
```

What’s wrong?

### Outer Join Application

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

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

```sql
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
Oh no…even more SQL

- Finally, updates/modifications to database
- INSERT, DELETE and UPDATE can be result of queries!

**INSERT**

- Give all customers with a Loan at Downtown branch a $200 savings account with same account number as Loan number

```sql
INSERT INTO Deposit
SELECT CustID, Loan-number, Branch-name, 200
FROM Loan
WHERE branch-name = 'Downtown';
```

**DELETE**

 Predicate in P can be as complex as any select clause
Delete all accounts located in New York

```sql
DELETE Deposit
WHERE branchname in (SELECT branchname
FROM Branch
WHERE branchcity='New York');
```

**How about this query?**

```sql
DELETE Deposit
WHERE balance < (SELECT avg(balance)
FROM Deposit);
```
Delete anomalies
- If delete/update request contains embedded select (sub-query) that references relation where deletions/update take place
- SQL standard disallows such requests
  - Alternate implementation: mark tuples in first round, and actual delete in second round

INSERT
- Can insert tuple with specified values
- Can insert set of tuples resulting from query

UPDATE
- Change a value in tuple without changing all values in the tuple
- Can update set of tuples by using query to select the set

Now, we are done.....kind of!
- More components to SQL:
  - Views
  - Constraints, Triggers
  - ...will get to these!