EECS-317 Data Management and Information Processing

Lecture 9 – Midterm Review

Steve Tarzia

Spring 2019

Northwestern

Announcements

- Midterm is next class (Thurs May 2nd)
  - Covers SQL queries.
    - Lectures 1-6
    - All homeworks
  - Open book, open notes, but you cannot share any materials.
  - Seats will be assigned.
  - Ten page practice exam (with answers) is posted.
  - Last year’s midterm (with answers) is also posted.
  - Don’t forget to do the practice homeworks in Canvas.

Why use a relational database?

- **Scalability** – work with data larger than computer's RAM
- **Persistence** – keep data around after your program finishes
- **Indexing** – efficiently sort & search along various dimensions
- **Integrity** – restrict data type, disallow duplicate entries
- **Deduplication** – save space, keep common data consistent
- **Concurrency** – multiple users or applications can read/write
- **Security** – different users can have access to specific data
- **"Researchability"** – SQL allows you to concisely express analysis

Sometimes we start with one redundant table and break it down to reflect the logical components

staff					
id	***name***	***department***	***building***	***room***	***faxNumber***
11	Bob	Industrial Eng.	Tech	100	1-1000
20	Betsy	Computer Sci.	Ford	100	1-5003
21	Fran	Industrial Eng.	Tech	101	1-1000
22	Frank	Chemistry	Tech	102	1-1000
35	Sarah	Physics	Mudd	200	1-2005
40	Sam	Materials Sci.	Cook	10	1-3004
54	Pat	Computer Sci.	Ford	102	1-5003

This is called ***Normalization***

staff		
id	***name***	***department***
11	Bob	1
20	Betsy	2
21	Fran	1
22	Frank	4
35	Sarah	5
40	Sam	7
54	Pat	2

department		
id	***name***	***building***
1	Industrial Eng.	1
2	Computer Sci.	2
4	Chemistry	1
5	Physics	4
7	Materials Sci.	5

building		
id	***name***	***faxNumber***
1	Tech	1-1000
2	Ford	1-5003
4	Mudd	1-2005
5	Cook	1-3004
6	Garage	1-6001

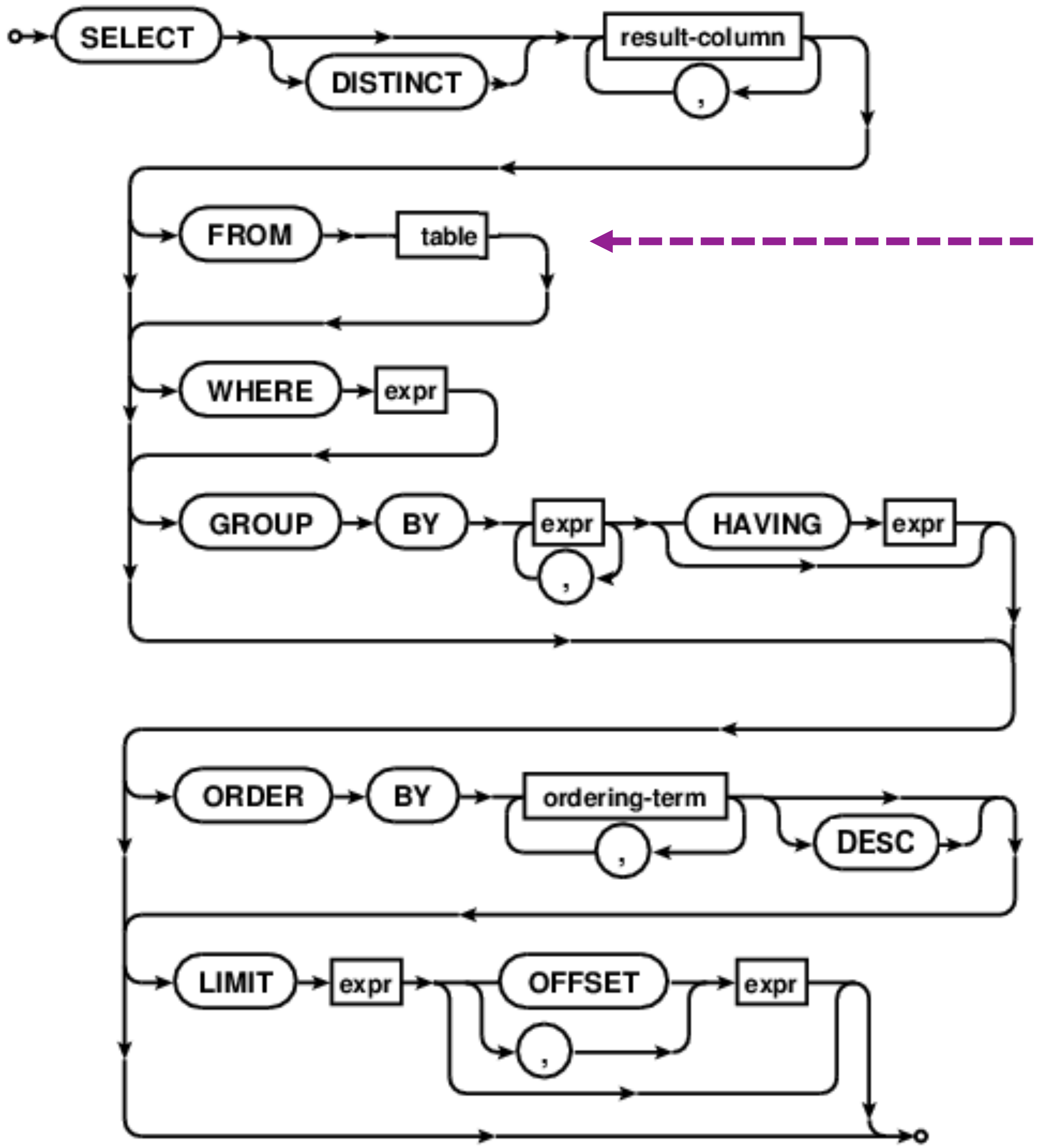
- Removes redundancy
  - Save space
  - Edit values in one place, so duplicates don't become inconsistent
- Tables can be populated separately
- **But**, you are adding a new *id* column for each table

Tables

- Represent objects, events, or relationships
  - Each of its rows must be uniquely identifiable
  - Has attributes that the DB will store in columns
  - Can refer to rows in other tables
- ***Objects***: people, places, or things
- ***Events***: usually associated with a specific time. Can recur.
- ***Relationships***: associations

Designing a set of tables is called *data modelling*, and it's best learned by example.

Basic SELECT Syntax



"table" can be:

- A single table
- Multiple tables JOINed together
- A subquery that returns a table

SELECT steps (abbreviated)

1. `FROM` chooses the table of interest
2. `WHERE` throws out irrelevant rows
3. `GROUP BY` identifies rows to combine
4. `SELECT` tells what values to return (allowing math and aggregation)
5. `HAVING` throws out irrelevant rows (after aggregation)
6. `ORDER BY` sorts
7. `LIMIT` throws out rows based on their position in the results

Each step gets closer to the specific result you want.

Integer vs. *floating point* division

- Computers store numbers in two basic ways:
  - **Integers** are whole numbers (0, 3, -40,921)
  - **Floating Point** numbers (*floats*) can be fractional (1.234, 0.0, -9.9×10^{-4})
- When doing arithmetic on two integers, an integer is always produced.
  - 1+1 = 2, 2-1=1, 4*3=12, **13/4=3**
- When doing arithmetic involving at least one float, a float is produced.
  - 1.0 + 1.0 = 2.0, 1.5 * 2 = 3.0, 13/4.0=3.25
- *Integer division is weird* – it always rounds down: **2/3 = 0**, **-5/2 = -3**
- Usually you need floating-point (not integer) division in your queries.
  - Just precede the expression with a floating point operation to force the division to be floating point: **1.0 * -5 / 2 = -2.5**

Aggregation functions

- COUNT, SUM, MIN, MAX, AVG
- These can be used to print out values that depend on multiple rows.
- For example, how many ounces of ingredients are used?
  - We have to add up the "Amount" from many rows to get this answer:

```
SELECT SUM(Amount) FROM Recipe_Ingredients
  WHERE MeasureAmountID=1;
```

  - ("ounce" corresponds to MeasureAmountID=1)

- `GROUP BY` causes aggregations to occur on subsets of rows, where rows are grouped according to some rule.
  - Each group contains rows having the same value for the grouping expression

```
SELECT SUM(Amount) FROM Recipe_Ingredients
  GROUP BY MeasureAmountID;
```

  - Same as above, but list amounts of all ingredients

GROUP_CONCAT() is another aggregator

- Concatenates non-null values, optionally with a separator string.
- *Eg.*: Print all the products in each category

```
SELECT CategoryDescription, GROUP_CONCAT(ProductName, ", ")
FROM Products NATURAL JOIN Categories GROUP BY CategoryID;
```

	CategoryDescription	GROUP_CONCAT(ProductName, ", ")
1	Accessories	Dog Ear Cyclecomputer, Dog Ear Helmet Mount Mirrors, Viscount C-500 Wireless Bike Computer, Kryptonite Advanced ...
2	Bikes	Trek 9000 Mountain Bike, Eagle FS-3 Mountain Bike, Viscount Mountain Bike, GT RTS-2 Mountain Bike
3	Clothing	Ultra-Pro Rain Jacket, StaDry Cycling Pants, Kool-Breeze Rocket Top Jersey, Wonder Wool Cycle Socks
4	Components	Victoria Pro All Weather Tires, Shinoman 105 SC Brakes, Shinoman Dura-Ace Headset, Eagle SA-120 Clipless Pedals, Pr...
5	Car racks	Road Warrior Hitch Pack, Ultimate Export 2G Car Rack
6	Wheels	X-Pro All Weather Tires, Turbo Twin Tires, Ultra-2K Competition Tire

GROUP BY explained

- GROUP BY combines multiple rows into one row in the result.
- Rows with the same value for the *grouping criterion* are grouped.
- An aggregation function is usually applied.

```
SELECT CategoryID, COUNT(*) AS category_count,
       MAX(RetailPrice) AS most_expensive_price
  FROM Products GROUP BY CategoryID;
```

ProductName	RetailPrice	CategoryID
Trek 9000 Mountain Bike	1200	2
Eagle FS-3 Mountain Bike	1800	2
Dog Ear Cyclecomputer	75	1
Victoria Pro All Weather Tires	54.95	4
Dog Ear Helmet Mount Mirrors	7.45	1
Viscount Mountain Bike	635	2
Viscount C-500 Wireless Bike Computer	49	1

CategoryID	category_count	most_expensive
1	18	179
2	4	1800
3	4	85
4	9	189
5	2	180
6	3	34

Subqueries

- Any single value, list of values, or table can be replaced by a subquery
- A **subquery** is a query that appears inside of parentheses.
  - The subquery is computed first and its result is "plugged into" the parent expression.

```
SELECT SUM(Amount) FROM Recipe_Ingredients
  WHERE MeasureAmountID=
    (SELECT MeasureAmountID FROM Measurements
     WHERE MeasurementDescription="Ounce");
```

INNER JOIN review

staff			
id	***name***	***room***	***departmentId***
11	Bob	100	1
20	Betsy	100	2
21	Fran	101	1

department		
id	***name***	***buildingId***
1	Industrial Eng.	1
2	Computer Sci.	2
4	Chemistry	1
1	Physics	4
1	Materials Sci.	5

```
SELECT * FROM staff JOIN department
ON staff.departmentId=department.id
```

In output,

- multiple matches leads to multiple rows.
- no matches leads to no rows

staff.***id***	staff.***name***	***staff.room***	staff.***departmentId***	department.***id***	department.***name***	department.***buildingId***
11	Bob	100	1	1	Industrial Eng.	1
11	Bob	100	1	1	Physics	4
11	Bob	100	1	1	Materials Sci.	5
20	Betsy	100	2	2	Computer Sci.	2
21	Fran	101	1	1	Industrial Eng.	1
21	Fran	101	1	1	Physics	4
21	Fran	101	1	1	Materials Sci.	5

NATURAL JOIN

- A shorthand notation to make some JOINs shorter to express.
- NATURAL JOIN matches rows using whatever columns have identical names.

For example:

```
SELECT * FROM Orders JOIN Order_Details
  ON Orders.OrderNumber=Order_Details.OrderNumber;
```

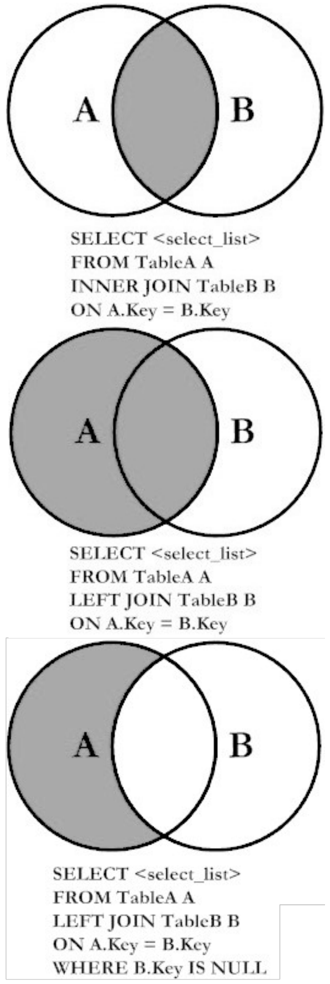
Becomes:

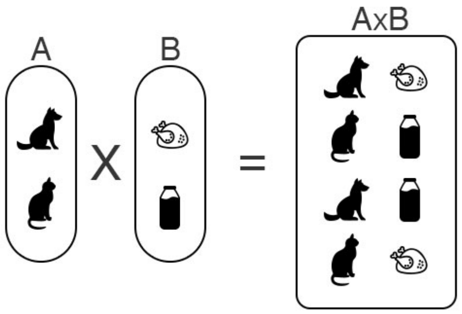
```
SELECT * FROM Orders NATURAL JOIN Order_Details;
```

Outer and Cross Joins

Introduced different types of JOINs:

- **INNER** (default): prints all pairs of rows (one from first table, one from second table) that satisfy the *JOIN predicate.*
- **LEFT**: same as INNER, but adds rows from LEFT table that never satisfied the JOIN predicate.
- **LEFT with exclusion**: only print rows from left table that never satisfied the JOIN predicate.
- **CROSS JOIN**: print the cartesian project, meaning all rows from the first table combined with all rows from the second table. There is no "ON" to match rows.





Cartesian Product of Two Sets.

staff			
id	***name***	***room***	***departmentId***
11	Bob	100	1
20	Betsy	100	*NULL*
21	Fran	101	1
22	Frank	102	99999
35	Sarah	200	5
40	Sam	10	7
54	Pat	102	2

department		
id	***name***	***buildingId***
1	Industrial Eng.	1
2	Computer Sci.	2
5	Physics	4
7	Materials Sci.	5

- Betsy and Frank have NULLs in the right haft of the output because no matching department was found.
- In other words no pair of rows was found to satisfy the ON staff.departmentId=department.id

```
SELECT * FROM staff LEFT JOIN department ON staff.departmentId=department.id;
```

staff.***id***	staff.***name***	***staff.room***	staff.***departmentId***	department.***id***	department.***name***	department.***buildingId***
11	Bob	100	1	1	Industrial Eng.	1
20	Betsy	100	*NULL*	***NULL***	***NULL***	***NULL***
21	Fran	101	1	1	Industrial Eng.	1
22	Frank	102	99999	***NULL***	***NULL***	***NULL***
35	Sarah	200	5	5	Physics	4
40	Sam	10	7	7	Materials Sci.	5
54	Pat	102	2	2	Computer Sci.	2

LEFT JOIN with Grouping

- When computing an *aggregation* on a *many-to-one* relationship, LEFT JOIN includes rows from the parent table with no children.

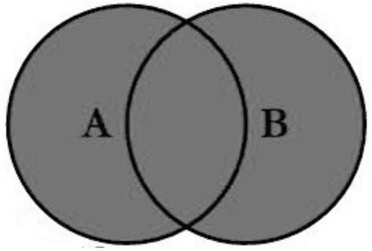
In ClassScheduling.slite, count the classes taught by each faculty member:

- If you want this report to include faculty members teaching zero classes, you must use LEFT JOIN:

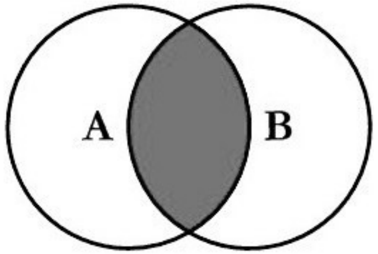
```
SELECT StaffID, ClassID,
    COUNT(ClassID) AS num_classes
  FROM Faculty NATURAL LEFT JOIN Faculty_Classes
  GROUP BY StaffID;
```

- Note that "`COUNT(*)`" would return "1" for faculty members with no classes, because there would still be one unmatched row from the left table.

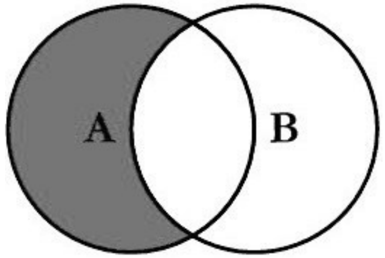
UNION, INTERSECT, and EXCEPT
are used to combine two SELECT statements



- **UNION** prints rows from *either of two* SELECTs (printing duplicates just once)



- **INTERSECT** prints rows *present in both* SELECTs

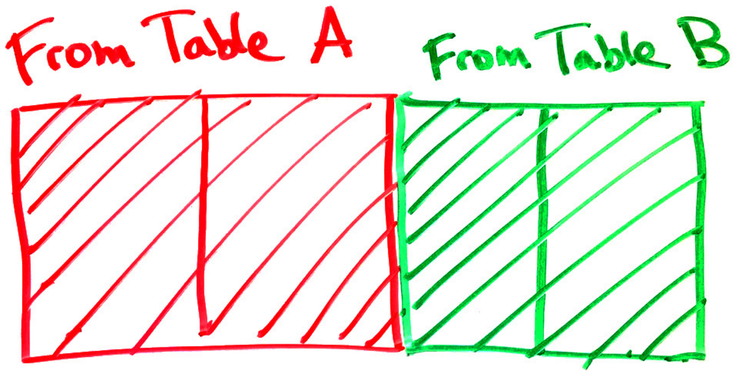


- **EXCEPT** prints rows *present in one* SELECT but *missing from another* SELECT

JOIN vs. UNION

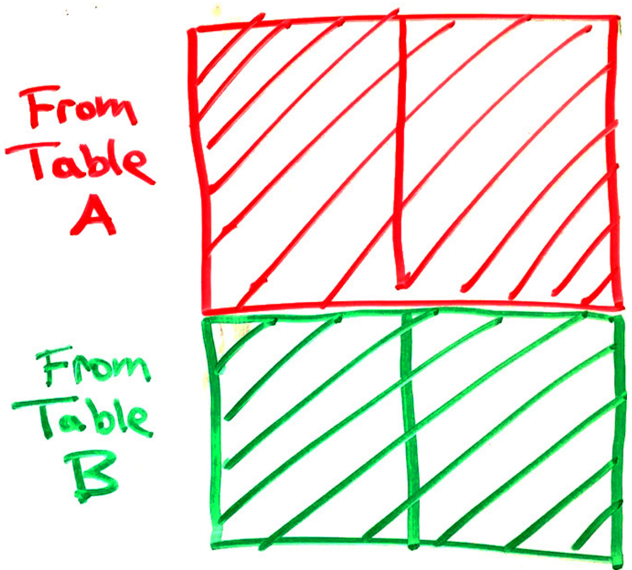
- JOINs combine tables *horizontally*.
  - Match rows from two tables based on one or more columns matching.
  - Creates a wider set of rows, adding **columns** from both tables.

JOIN:



- UNION, INTERSECT, and EXCEPT combine result tables
  - Number & type of columns in the two result tables must match
  - Changes the number of **rows**, not columns

UNION:



Summing an indicator variable

Two ways to count recipes with "salsa" in description:

- `SELECT COUNT(*) FROM Recipes WHERE` **`RecipeTitle LIKE "%salsa%";`**
  - `WHERE` clause keeps just the rows matching "salsa," then these rows are counted.
- `SELECT SUM(`**`RecipeTitle LIKE "%salsa%"`**`) FROM Recipes;`
  - A column is created for every recipe indicating whether its title matches "salsa" or not.
  - Column's value will be **1** if it matches and **0** if not.
  - Sum of all the ones and zeros will be the count of matching recipes.
- First approach is easier to understand, but second is shorter.

CASE gives if-then-else behavior

`WHEN` condition is tested for every row, giving *true* or *false*

```
SELECT CASE WHEN CategoryID=2
       THEN  "Bike"
       ELSE  ProductName  END FROM Products;
```

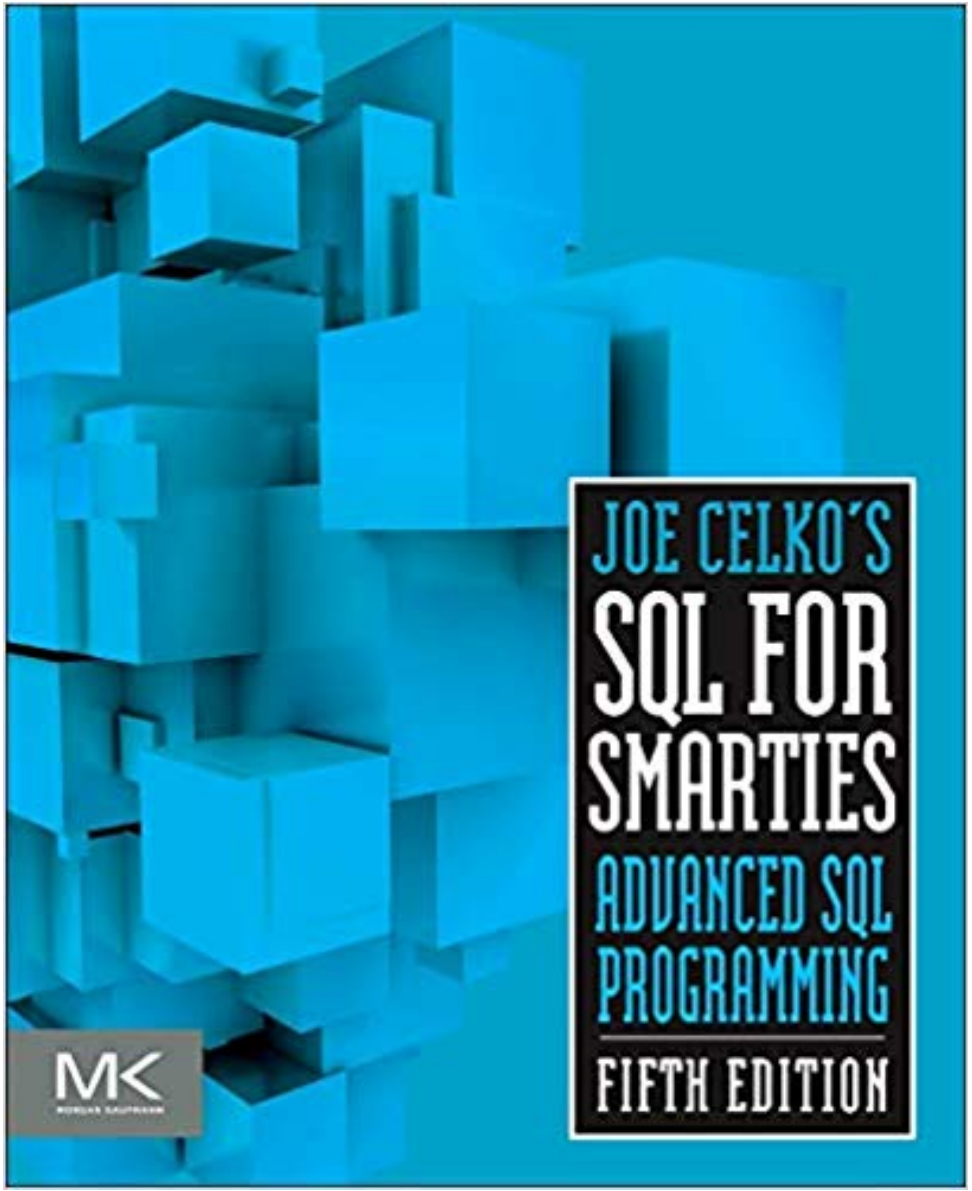
If condition is *true* then use the first value.

If condition is *false* then use the second value.

Output:

1	Bike
2	Bike
3	Dog Ear Cyclecomputer
4	Victoria Pro All Weather Tires
5	Dog Ear Helmet Mount Mirrors
6	Bike
7	Viscount C-500 Wireless Bike Computer
8	Kryptonite Advanced 2000 U-Lock
9	Nikoma Lok-Tight U-Lock

If you wish to dig further into SQL



What's coming in the second half of the course?

- More data modeling (designing new databases)
- Indexing to handle large databases
- Defining databases and adding data
- Numeric formats
  - integers, floats, precision
  - Dates and times
- Text encodings
  - ASCII, UTF-8, special characters
- Organizing data in files (semi-structured data)
  - CSV, XML, JSON
- Messy data
  - Missing entries, fuzzy matching
- ... and perhaps more, time permitting

HW3 Q4

```
SELECT DISTINCT Order_Details.OrderNumber
FROM Order_Details NATURAL JOIN Products NATURAL JOIN Product_Vendors
 JOIN Order_Details AS od2 ON Order_Details.OrderNumber=od2.OrderNumber
GROUP BY Order_Details.OrderNumber, VendorID
HAVING COUNT(DISTINCT Order_Details.ProductNumber)=COUNT(DISTINCT
od2.ProductNumber)
```

HW3 Q5 strategy

Orders must be fulfilled by 2 vendors.

= All orders − orders that can be fulfilled by 1 vendor

− orders that can be fulfilled by ≥ 3 vendors

Q4

EXCEPT = subtraction

= there are 3 different products in the order available from 3 different vendors

HW3 Q5

```
-- Start with all orders

SELECT OrderNumber FROM Orders


EXCEPT


-- remove the orders that can be satisfied with one vendor (Q4):

SELECT DISTINCT Order_Details.OrderNumber

FROM Order_Details NATURAL JOIN Products NATURAL JOIN Product_Vendors

 JOIN Order_Details AS od2 ON Order_Details.OrderNumber=od2.OrderNumber

GROUP BY Order_Details.OrderNumber, VendorID

HAVING COUNT(DISTINCT Order_Details.ProductNumber)=COUNT(DISTINCT od2.ProductNumber)

EXCEPT


-- remove the orders that have three different products available from three different vendors

SELECT o1.OrderNumber

-- list each possible combination of three line items from each order

FROM Order_Details AS o1 JOIN Order_Details AS o2 ON o1.OrderNumber=o2.OrderNumber

JOIN Order_Details AS o3 ON o3.OrderNumber=o1.OrderNumber

JOIN Product_Vendors v1 ON o1.ProductNumber=v1.ProductNumber

JOIN Product_Vendors v2 ON o2.ProductNumber=v2.ProductNumber

JOIN Product_Vendors v3 ON o3.ProductNumber=v3.ProductNumber

WHERE

-- vendors are different

v1.VendorID != v2.VendorID

AND v1.VendorID != v3.VendorID

AND v2.VendorID != v3.VendorID

-- products are different

AND o1.ProductNumber != o2.ProductNumber

AND o1.ProductNumber != o3.ProductNumber

AND o2.ProductNumber != o3.ProductNumber;
```