EECS-317 Data Management and Information Processing

Lecture 2 – Structured Query Language (SQL)

Steve Tarzia

Spring 2019

Northwestern

Please take a copy of the 3 handouts before sitting down

Announcements

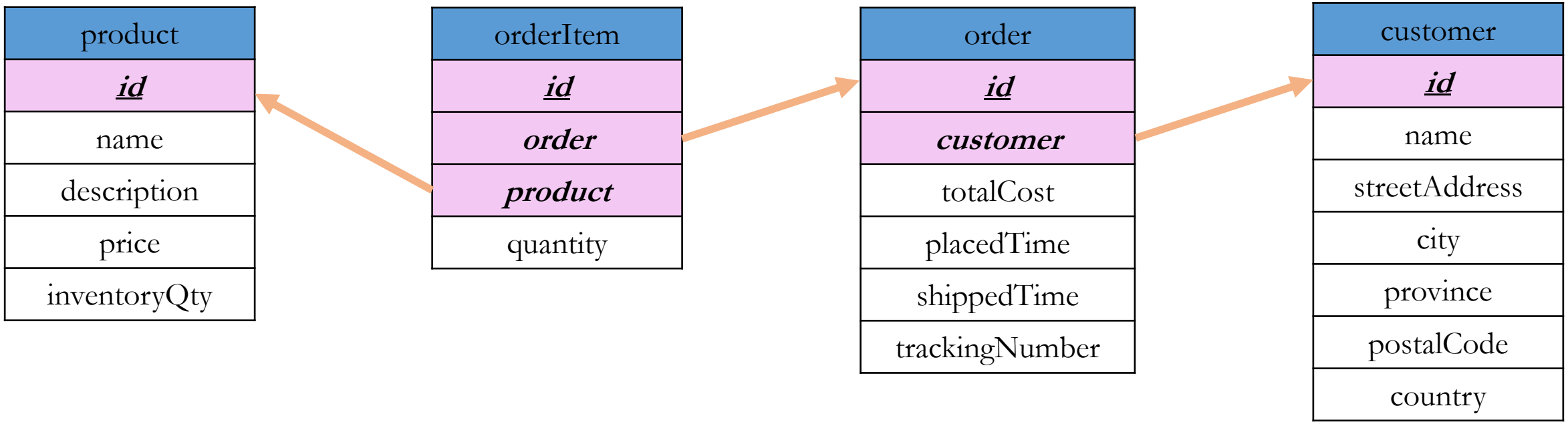
- First HW assignment was posted, due April 15th (a Monday).
- First "reading" assignment was also posted:
  - Complete Datacamp's Intro to SQL for Data Science.
  - You actually should do this first, before the homework.
- Office hours will be held in the Wilkinson Lab (Tech M338)

Last lecture: Data Modeling with Tables

- *Complex data* are more than just streams of numbers!
- *Data model* or *schema* defines the data's structure
  - It's a list of *tables*, each with a fixed number of *columns*
  - Data *rows* are added after the data model is designed.
    - Within a row, each column may store only one value.
    - A column may refer to a row in another table.
- These are called *Relational* or *SQL* databases.
  - Can represent much more complex data than a simple spreadsheet.
  - Eliminate data redundancy:
    Saves space and allows updates to happen in one place
  - Allows objects, events, and relationships to be added separately

Last lecture (continued)

- Introduced data model diagrams, but did not go into much detail.



DB Browser for SQLite

Structured Query Language (SQL)

- The standard programming language for relational databases
- Each DB Management System (DBMS) has its own dialect
  - In this course we will be using SQLite's and MySQL's variants of SQL
- SQL is a **declarative** language (most other languages are imperative)
  - You describe the results you want to see
  - You do not describe the detailed steps necessary to gather those results
  - The DBMS cleverly determines an **execution plan** behind the scenes to carry out your requested analysis.
- We will be using a client program to connect to the DBMS and running SQL statements **interactively**:
  - run one statement and look at the results before running another one

SELECT gets data

```
SELECT FirstName, LastName FROM customers WHERE City = "Paris";
```

Columns to print | Table to examine | Filter

Result is a table with two rows:

FirstName	*LastName*
Camille	Bernard
Dominique	Lefebvre

Filtering, sorting, and limiting

We can use more complex filters:

```
SELECT FirstName, LastName FROM customers
     WHERE City = "Chicago"
                AND (State = "Illinois"
                       OR State = "IL");
```

Get all columns, sort the results (descending) and limit the results to just the first ten rows:

```
SELECT * FROM tracks ORDER BY UnitPrice DESC LIMIT 10;
```

Arithmetic

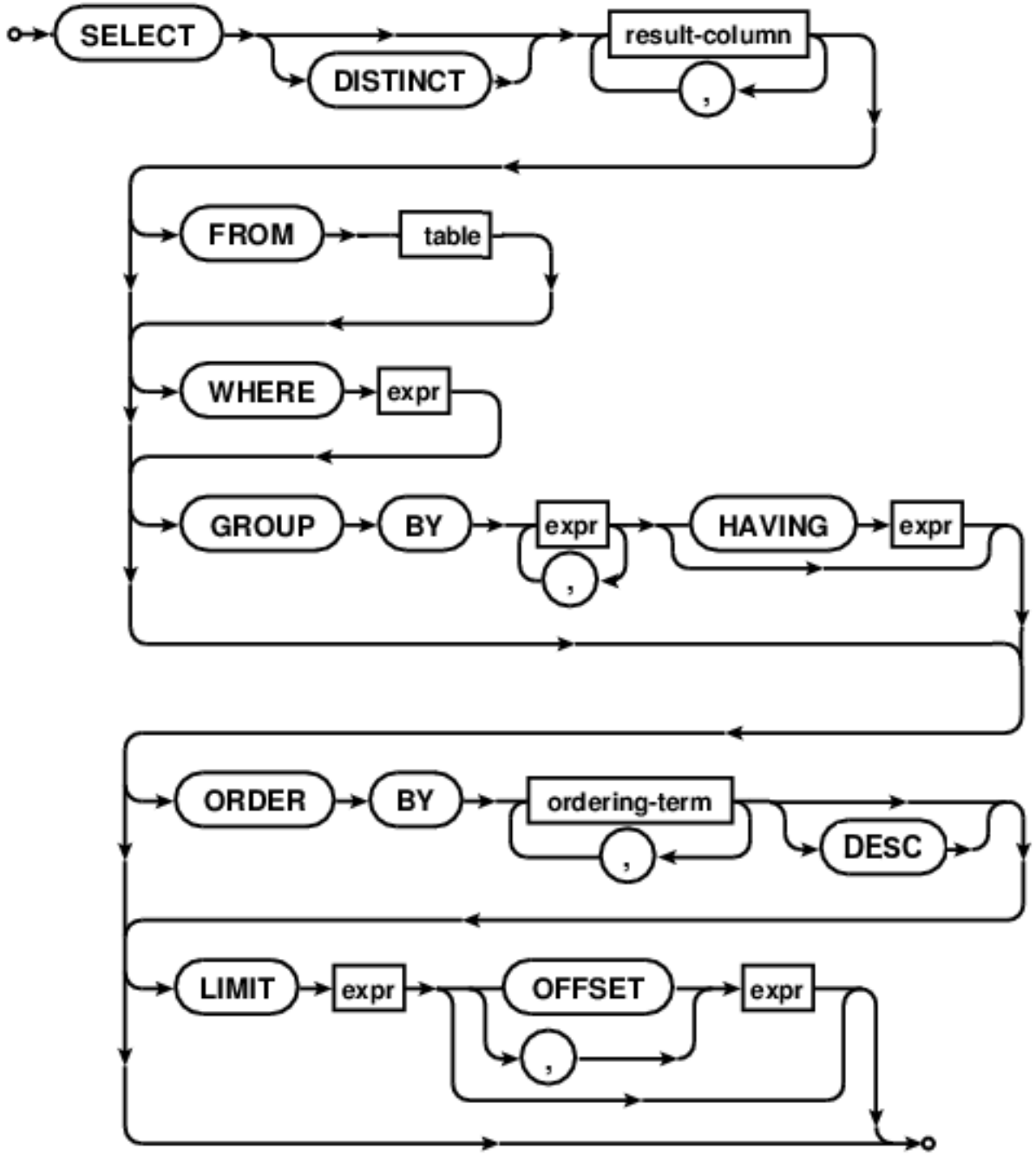
Your SELECT statements can include arithmetic

```
SELECT 1+1;
SELECT ABS(COS(PI()));
SELECT Name, UnitPrice / (Milliseconds/1000/60)
    AS PricePerMinute FROM tracks;
```

Check your DBMS's documentation for the specific math functions.

Syntax diagrams

- Any path from start to end is a valid statement.
- Choose which arrows to follow
- The rectangles refer to other diagrams.
- Used by our SQL book
- Used by SQLite online docs: https://sqlite.org/lang.html



Syntax grammars

- A set of rules for building all possible statements
- Used by MySQL docs
- Optional items are in square braces: `[ ]`
- Pipe character for "or": `this | that`
- Curly braces for a required choice: `{one | two}`
- `...` for repetition
- Lowercase italics for things defined elsewhere.

```
SELECT
  [ALL | DISTINCT | DISTINCTROW ]
    [HIGH_PRIORITY]
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT]
    [SQL_BUFFER_RESULT]
    [SQL_CACHE | SQL_NO_CACHE][SQL_CALC_FOUND_ROWS]
  select_expr [, select_expr ...]
  [FROM table_references
    [PARTITION partition_list]]
  [WHERE where_condition]
  [GROUP BY {col_name | expr | position}
    [ASC | DESC], ... [WITH ROLLUP]]
  [HAVING where_condition]
  [ORDER BY {col_name | expr | position}
    [ASC | DESC], ...]
  [LIMIT {[offset,] row_count | row_count OFFSET offset}]
  [PROCEDURE procedure_name(argument_list)]
  [INTO OUTFILE 'file_name'
      [CHARACTER SET charset_name]
      export_options
    | INTO DUMPFILE 'file_name'
    | INTO var_name [, var_name]]
  [FOR UPDATE | LOCK IN SHARE MODE]]
```

Grouping

The GROUP BY clause combines multiple rows and lets you perform aggregation math functions.

```
SELECT AlbumId,
    SUM(Milliseconds/1000/60) AS AlbumMinutes
  FROM tracks GROUP BY AlbumId ORDER BY AlbumMinutes;
```

Result:

AlbumId	*AlbumMinutes*
340	0.86300000
345	1.11065000
318	1.68821667
…	…

SQLite

- A lightweight and easy-to-setup database management system
- Similar to Microsoft Access, but free and more portable
- Can handle very large databases (terabytes)
- The whole database is stored in a single file (.db or .sqlite)
- But SQLite *does not* handle remote access from multiple users

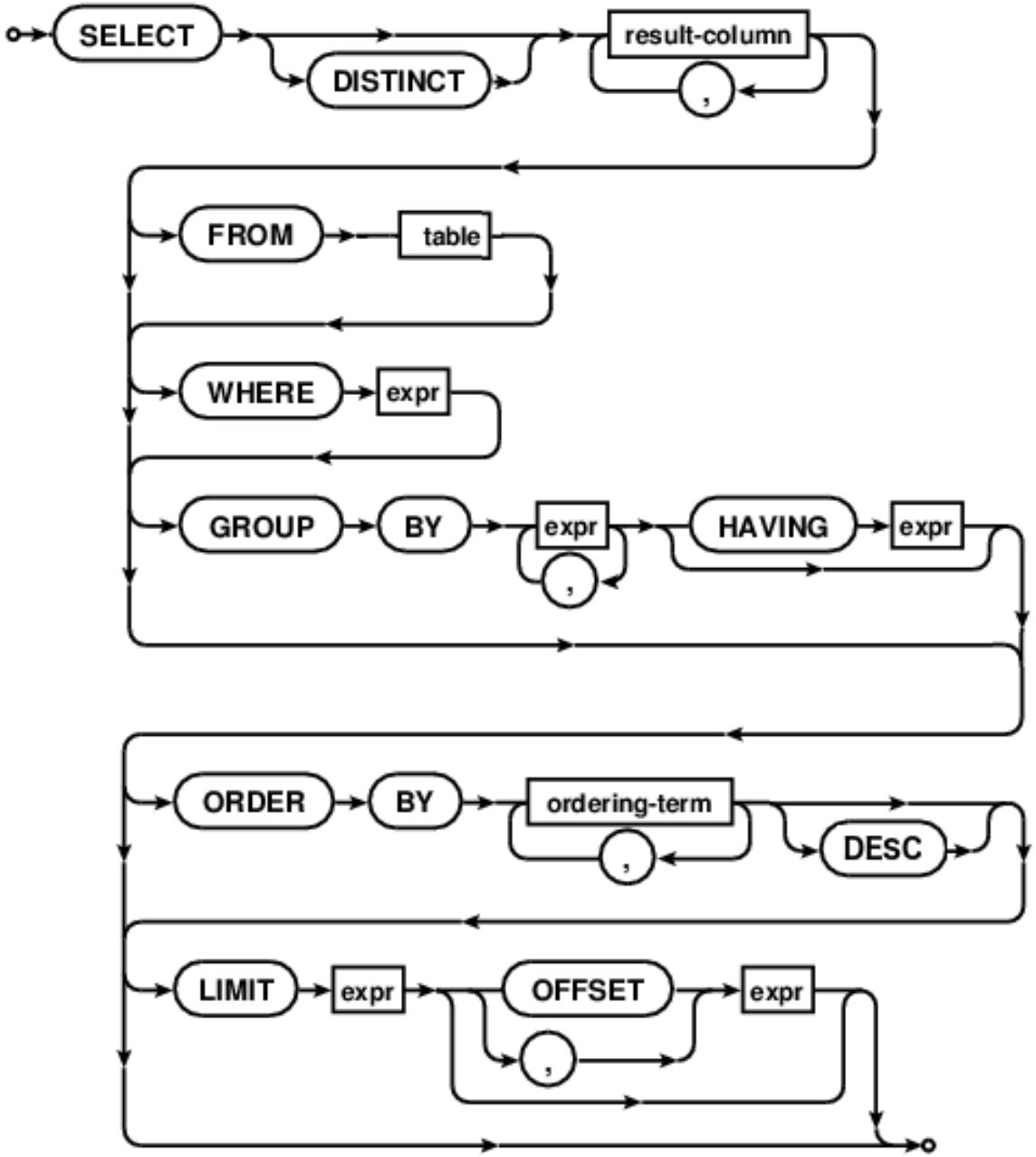
A good choice for an individual needing to set up his/her own database.

- Download a version of it from sqlitebrowser.org

SQLite SELECT Syntax

For example:

```
SELECT FirstName, LastName
FROM customers WHERE City =
"Paris";
```



SELECT queries are series of *filtering* & *manipulation* steps

1. The `FROM` expression gives the starting point – a full table.
   The final result will be a subset or aggregation of this.
2. The `WHERE` expression keeps only those rows passing some test
   This expression can be very complex, but it must be something than can be evaluated on each row, one at a time.
3. `GROUP BY` combines rows if something about them is the same
4. The `SELECT` result-columns are computed, including aggregation.
   At this point we have thrown out the columns we don't need.
5. `HAVING` expression keeps only the aggregated rows passing a test.
6. `ORDER BY` sorts what's left.
7. `LIMIT` truncates the results to just a certain number of rows.

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.

What's the average price of a bike car rack?

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

Products table has the price info, so we start there:

```
SELECT * FROM Products
```

This placeholder will change in step 4.

ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
Filter	Filter	Filter	Filter	Filter	Filter
1	Trek 9000 Mountain Bike	NULL	1200	6	2
2	Eagle FS-3 Mountain Bike	NULL	1800	8	2
3	Dog Ear Cyclecomputer	NULL	75	20	1
4	Victoria Pro All Weather Tires	NULL	54.95	20	4
5	Dog Ear Helmet Mount Mirrors	NULL	7.45	12	1
6	Viscount Mountain Bike	NULL	635	5	2
7	Viscount C-500 Wireless Bike Computer	NULL	49	30	1
8	Kryptonite Advanced 2000 U-Lock	NULL	50	20	1
9	Nikoma Lok-Tight U-Lock	NULL	33	12	1
10	Viscount Microshell Helmet	NULL	36	20	1

What's the average price of a bike car rack?

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

We only need the bike rack products, so we filter on CategoryID = 5

```
SELECT * FROM Products
  WHERE CategoryID = 5;
```

ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
39	Road Warrior Hitch Pack	*NULL*	175	6	5
40	Ultimate Export 2G Car Rack	*NULL*	180	8	5

What's the average price of a bike car rack?

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

A `GROUP BY` statement is not needed because we will group *all* of the rows together.

```
SELECT * FROM Products
  WHERE CategoryID = 5
```

ProductNumber	ProductName	ProductDescription	RetailPrice	QuantityOnHand	CategoryID
39	Road Warrior Hitch Pack	*NULL*	175	6	5
40	Ultimate Export 2G Car Rack	*NULL*	180	8	5

What's the average price of a bike car rack?

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

We want the RetailPrice column, and we want to aggregate all the rows with the average function

```
SELECT
   AVG(RetailPrice)
   FROM Products WHERE
   CategoryID = 5
```

AVG(RetailPrice)
177.5

Recipes.sqlite (download it from Canvas)

- Print an alphabetically sorted list of ingredients (hint: ORDER BY).
- How many times is butter used as an ingredient?
- How many ingredients are in the Yorkshire Pudding recipe?
- What percentage of ingredients are vegetarian? Vegan?
- How many recipes have multi-word names? Nine-letter names?

Print an alphabetically sorted list of ingredients (hint: ORDER BY).

How many times is butter used as an ingredient?

How many ingredients are in the Yorkshire Pudding recipe?

What percentage of ingredients are vegetarian?

How many recipes have multi-word names? Nine-letter names?

Recipes.sqlite (answers)

- Print an alphabetically sorted list of ingredients (hint: ORDER BY).

```
SELECT IngredientName FROM Ingredients ORDER BY IngredientName;
```

- How many times is butter used as an ingredient?

```
SELECT COUNT(*) FROM Recipe_Ingredients WHERE IngredientID=47;
```

- How many ingredients are in the Yorkshire Pudding recipe?

```
SELECT COUNT(*) FROM Recipe_Ingredients WHERE RecipeID=10;
```

- What percentage of ingredients are vegetarian?

```
SELECT 100.0* COUNT(*)/(SELECT COUNT(*) FROM Ingredients)
FROM Ingredients WHERE IngredientClassID NOT IN (2, 10);
```

- How many recipes have multi-word names? Nine-letter names?

```
SELECT COUNT(*) FROM Recipes WHERE RecipeTitle LIKE "% %";
SELECT COUNT(*) FROM Recipes WHERE RecipeTitle LIKE "_________";
SELECT COUNT(*) FROM Recipes WHERE LENGTH(RecipeTitle) = 9;
```

Recap: SQL Basics

- Showed syntax diagram for SELECT.
- Described my 7-step process for building a SELECT query.
  - Start with a short query:

    ```
    SELECT * FROM some_table
    ```

  - Gradually refine the results, making the query more complex.
  - Choose table, filter, choose columns, apply mathematic operations, sort, etc.
- Demonstrated how to build a query to answer a few questions about recipes.