30 May 2015

General Assembly -- Boston

SQL Bootcamp

Let's run some queries

Starting simple

```
SELECT
    au_id, au_fname, au_lname
FROM
    books.authors
;

```

- Gives us selected fields for all records

```
+-------+-----------+-------------+
| au_id | au_fname  | au_lname    |
+-------+-----------+-------------+
| A01   | Sarah     | Buchman     |
| A02   | Wendy     | Heydemark   |
| A03   | Hallie    | Hull        |
| A04   | Klee      | Hull        |
| A05   | Christian | Kells       |
| A06   |           | Kellsey     |
| A07   | Paddy     | O'Furniture |
+-------+-----------+-------------+

```

Get different fields

```
SELECT
    au_id, au_lname, city, state
FROM
    books.authors
;

```

Gives us a different selection of fields, again for all records

```
+-------+-------------+---------------+-------+
| au_id | au_lname    | city          | state |
+-------+-------------+---------------+-------+
| A01   | Buchman     | Bronx         | NY    |
| A02   | Heydemark   | Boulder       | CO    |
| A03   | Hull        | San Francisco | CA    |
| A04   | Hull        | San Francisco | CA    |
| A05   | Kells       | New York      | NY    |
| A06   | Kellsey     | Palo Alto     | CA    |
| A07   | O'Furniture | Sarasota      | FL    |
+-------+-------------+---------------+-------+

```

Simple Query

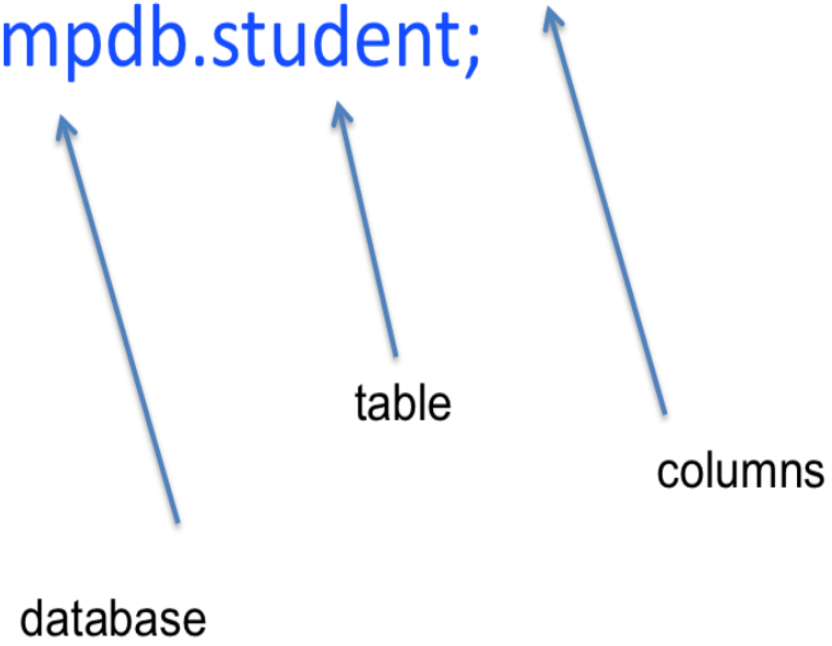
```
SELECT
    name, sex, student_id
FROM
    sampdb.student
LIMIT 5
;

```

- Query gets all fields of all records in table
- LIMIT, however, truncates output -- useful when we want to peek, but don't want the whole table

```
+--------+-----+------------+
| name   | sex | student_id |
+--------+-----+------------+
| Megan  | F   |          1 |
| Joseph | M   |          2 |
| Kyle   | M   |          3 |
| Katie  | F   |          4 |
| Abby   | F   |          5 |
+--------+-----+------------+

```

Simple Query

```
SELECT name, sex, student_id
FROM sampdb.student;
```



We could also use:
SELECT * FROM sampdb.student
- << * >> is a wildcard meaning “all columns”

Let's unpack that:

"database"

A program concept for holding "tables"

Comparable to a folder in your computer's file system

"table"

A program concept holding data about instances of a particular entity type - students, presidents, exams

These data are held in "records" or "rows"

"record"

Data about a particular instance eg, particular student or particular movie

Data is in a collection of fields or columns

Comparable to a row in a spreadsheet

"Column" / "field"

An attribute of an entity type about which data is collected - a student's name, or sex, or the state of origin of a president

Comparable to a cell in a spreadsheet row

Schema Exploration

How do we see what is available to us?

What databases are available?

```
SHOW DATABASES;

```

```
+------------------------+
| Database               |
+------------------------+
| information_schema     |
| books                  |
| csv_load               |
| demo_app_db            |
| entities               |
| google_api             |
| homework               |
| join_sample            |
| learning               |
| mta                    |
| mysql                  |
| sampdb                 |
+------------------------+

```

- Shows the databases available to us on the server

What tables are available?

```
SHOW TABLES IN books;

```

```
+-----------------+
| Tables_in_books |
+-----------------+
| au_orders       |
| authors         |
| dups            |
| employees       |
| empsales        |
| hier            |
| publishers      |
| roadtrip        |
| royalties       |
| telephones      |
| temps           |
| time_series     |
| title_authors   |
| titles          |
+-----------------+

```

- Shows the tables available to us within a particular database

What columns are available?

```
DESCRIBE books.authors;
```

```
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| au_id    | char(3)     | NO   | PRI | NULL    |       |
| au_fname | varchar(15) | NO   |     | NULL    |       |
| au_lname | varchar(15) | NO   |     | NULL    |       |
| phone    | varchar(12) | YES  |     | NULL    |       |
| address  | varchar(20) | YES  |     | NULL    |       |
| city     | varchar(15) | YES  |     | NULL    |       |
| state    | char(2)     | YES  |     | NULL    |       |
| zip      | char(5)     | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

- Shows columns in the table, their data types, whether they are primary keys, whether they can take NULL values

We can see how a table is created

```
SHOW CREATE TABLE books.authors\G
```

```
*************************** 1. row ***************************
       Table: authors
Create Table: CREATE TABLE `authors` (
  `au_id` char(3) NOT NULL,
  `au_fname` varchar(15) NOT NULL,
  `au_lname` varchar(15) NOT NULL,
  `phone` varchar(12) DEFAULT NULL,
  `address` varchar(20) DEFAULT NULL,
  `city` varchar(15) DEFAULT NULL,
  `state` char(2) DEFAULT NULL,
  `zip` char(5) DEFAULT NULL,
  PRIMARY KEY (`au_id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1

```

- Shows columns in the table, their data types, whether they are primary keys, whether they can take NULL values

Practice

Find a table in books that has 'emp_id' as primary key

Find a table in books that has 'emp_name'

Write a query outputting emp_id and emp_name

What columns are available in sampdb.president?

Write a query outputting each President's first name, last name, date of birth and date of death

What is SQL?

Role of SQL in Data Analysis

1. **Data analysis requires:**

   Getting, cleaning, loading, 'modeling'

   Exploring, manipulating, extracting, analyzing

   Reporting, visualization

2. **SQL is best at analysis, but all help with all of this**

Getting and Cleaning

1. Something else has to get the data to CSV or database
2. SQL string functions can help clean and reformat data
3. Tricks with LOAD and INSERT can remodel data

Reporting and Visualization

1. **SQL has zero visualization capacity, but it can output data as needed for your graphics program**
2. **"Reporting" is here sort of a term of art**

   E.g. pivot tables, cross tabs, and so forth

   SQL can do an awful lot, but isn't great at this

   Implementations do provides base layers to reporting tools like Jasper, Pentaho, etc

Analysis and Transformation

1. **SQL's stock in trade is filtering, aggregating and grouping on set-based criteria**

   Filter WHERE records meet some stated criteria

2. **State equals Illinois, name in (Smith, Baker, Han)**
3. **Date before 2012-06-01**
4. **Score greater than 15**

   GROUP BY values in a field, or in a combination of fields

5. **Uses that capacity to combine information in different sets / tables by matching records on some criteria**
6. **We can nest these abilities into "subqueries", generating criteria on the fly**

As A Language - Application Usage

1. **I.e., why we care**
2. **A syntax for:**

   Describing tables

   Extracting and manipulating data from those tables

   Plus dealing with the cruft of managing data: loading, indexing, etc
3. **The action is in bullet #2, but we have to deal with the other stuff too**

Extracting and Manipulating

1. **Extracting:**

   We want to specify criteria for which records we display or analyze

   Combine data extracted from several tables

2. **Manipulating:**

   Transform values in fields

   Calculate values for fields of groups of records

3. **These fundamental operations can be extended to very complex transformations**

What is SQL -- Architecture

A "Standard", Not a Program

- **There are many SQLs**

  MySQL / Postgres / SQLite / Microsoft Access / Oracle / Sybase / etc etc etc

- **Each is an implementation of a standard**

  "SQL" is basically a program's promise to behave in a defined way in response to a particular query for a particular set of data

  "Client / Server"

- **Most implementations are actually two programs**

  One is a front end / client, giving us access to a server and showing its outputs

- **MySQL Workbench**

  The other is a back end / server, holding the data and processing the queries

- **MySQL**

  MySQL Workbench is to MySQL as Chrome browser is to an HTTP server

Client / Server Implications

1. ## We can access existing data

   We don't need data dumps to do analysis

2. ## We do need some sensible and safe access to the existing databases

   If your production database has a million records, you can analyze them all without downloading them all

3. ## We can access remote data

   If your servers are on Rackspace or Amazon you can still get at the data

4. ## Programs can access the data

   Languages like Ruby, Python, PHP all have tools for interacting with databases

   They act as clients, but getting data for (say) serving web pages rather than display to an analyst

Excel vs SQL


I asks for a brother
and dis is whats I gets.
ICANHASCHEEZBURGER.COM

Why Excel?

1. **Good graphic interface orients us to the data and the tools**

   We can find tools just by clicking around the menus

2. **Excel organizes models visually, so we can _see_ the relationships among cells**

   We can see the cells flowing into a formula

   We imagine extensions of our models by thinking through new visual relationships

3. **Excel handles many issues we don't even know about**

   Until we use a tool with less hand-holding - like SQL

Why SQL? -- Analytic

1. **Operated through text statements**

   Easy to identify and extend patterns

2. **Replicable**

   If a new query resembles a prior one, just copy and revise the prior

3. **Chainable**

   We can easily add conditions and "filters"

   We can write a query and refer to its results in another query

4. **Traceable**

   We can see how we got to a result by examining the query

5. **Non-visual modeling allows us to fluidly work in more than two dimensions**

Implications

1. **Excel gets used like a Swiss Army Knife**

   We have problems, Excel gives us path to solve them

2. **SQL can do the same things and more, BUT requires infrastructure, training and problem solving - i.e., brain damage**

   When we do need it, who has time for training?

Why SQL? -- Organizational

1. A lot of data already in your company's relational databases
2. Dev team familiar with SQL
3. SQL easier to automate
4. Easy to operate over a network

Manipulating Table Records

Behold, some data

Simple query

```
SELECT
    first_name,
    last_name,
    state
FROM
    sampdb.president
LIMIT
    20
;

```

- No order imposed on the results

Results

```
+---------------+------------+-------+
| first_name    | last_name  | state |
+---------------+------------+-------+
| John          | Adams      | MA    |
| John Quincy   | Adams      | MA    |
| Chester A.    | Arthur     | VT    |
| James         | Buchanan   | PA    |
| George H.W.   | Bush       | MA    |
| George W.     | Bush       | CT    |
| James E.      | Carter     | GA    |
| Grover        | Cleveland  | NJ    |
| William J.    | Clinton    | AR    |
| Calvin        | Coolidge   | VT    |
| Dwight D.     | Eisenhower | TX    |
| Millard       | Fillmore   | NY    |
| Gerald R.     | Ford       | NE    |
| James A.      | Garfield   | OH    |
| Ulysses S.    | Grant      | OH    |
| Warren G.     | Harding    | OH    |
| Benjamin      | Harrison   | OH    |
| William H.    | Harrison   | VA    |
| Rutherford B. | Hayes      | OH    |
| Herbert C.    | Hoover     | IA    |
+---------------+------------+-------+

```

We can specify an order

Re-order by state

```
SELECT
    first_name,
    last_name,
    state
FROM
    sampdb.president
ORDER BY
    state
LIMIT
    20
;

```

Results, Ordered

```
+-------------+-----------+-------+
| first_name  | last_name | state |
+-------------+-----------+-------+
| William J.  | Clinton   | AR    |
| Richard M.  | Nixon     | CA    |
| George W.   | Bush      | CT    |
| James E.    | Carter    | GA    |
| Herbert C.  | Hoover    | IA    |
| Ronald W.   | Reagan    | IL    |
| Barack      | Obama     | IL    |
| Abraham     | Lincoln   | KY    |
| George H.W. | Bush      | MA    |
| John        | Adams     | MA    |
| John F.     | Kennedy   | MA    |
| John Quincy | Adams     | MA    |
| Harry S     | Truman    | MO    |
| James K.    | Polk      | NC    |
| Andrew      | Johnson   | NC    |
| Gerald R.   | Ford      | NE    |
| Franklin    | Pierce    | NH    |
| Grover      | Cleveland | NJ    |
| Martin      | Van Buren | NY    |
| Franklin D. | Roosevelt | NY    |
+-------------+-----------+-------+

```

- Note that ordering applies only to RESULTS

We can re-order by non-string, non-alphabetized fields

Re-order by birth

```
SELECT
    first_name,
    last_name,
    birth
FROM
    sampdb.president
ORDER BY
    birth
LIMIT
    10
;

```

Results

```
+-------------+------------+------------+
| first_name  | last_name  | birth      |
+-------------+------------+------------+
| George      | Washington | 1732-02-22 |
| John        | Adams      | 1735-10-30 |
| Thomas      | Jefferson  | 1743-04-13 |
| James       | Madison    | 1751-03-16 |
| James       | Monroe     | 1758-04-28 |
| Andrew      | Jackson    | 1767-03-15 |
| John Quincy | Adams      | 1767-07-11 |
| William H.  | Harrison   | 1773-02-09 |
| Martin      | Van Buren  | 1782-12-05 |
| Zachary     | Taylor     | 1784-11-24 |
+-------------+------------+------------+

```

And we can reverse the order

Re-order by birth, reversed

```
SELECT
    first_name,
    last_name,
    birth
FROM
    sampdb.president
ORDER BY
    birth DESC
LIMIT
    10
;

```

Results

```
+-------------+------------+------------+
| first_name  | last_name  | birth      |
+-------------+------------+------------+
| William J.  | Clinton    | 1946-08-19 |
| George W.   | Bush       | 1946-07-06 |
| James E.    | Carter     | 1924-10-01 |
| George H.W. | Bush       | 1924-06-12 |
| John F.     | Kennedy    | 1917-05-29 |
| Gerald R.   | Ford       | 1913-07-14 |
| Richard M.  | Nixon      | 1913-01-09 |
| Ronald W.   | Reagan     | 1911-02-06 |
| Lyndon B.   | Johnson    | 1908-08-27 |
| Dwight D.   | Eisenhower | 1890-10-14 |
+-------------+------------+------------+

```

We can re-order by any field

Re-order by first_name

```
SELECT
    first_name,
    last_name,
    state
FROM
    sampdb.president
ORDER BY
    first_name
LIMIT
    10
;

```

Results

```
+-------------+------------+-------+
| first_name  | last_name  | state |
+-------------+------------+-------+
| Abraham     | Lincoln    | KY    |
| Andrew      | Johnson    | NC    |
| Andrew      | Jackson    | SC    |
| Benjamin    | Harrison   | OH    |
| Calvin      | Coolidge   | VT    |
| Chester A.  | Arthur     | VT    |
| Dwight D.   | Eisenhower | TX    |
| Franklin    | Pierce     | NH    |
| Franklin D. | Roosevelt  | NY    |
| George      | Washington | VA    |
+-------------+------------+-------+

```

We can re-order by several fields

Re-order by state and first_name

```
SELECT
    state,
    first_name,
    last_name
FROM
    sampdb.president
ORDER BY
    state,
    first_name
;

```

Results

```
+-------+---------------+------------+
| state | first_name    | last_name  |
+-------+---------------+------------+
| AR    | William J.    | Clinton    |
| CA    | Richard M.    | Nixon      |
| CT    | George W.     | Bush       |
| GA    | James E.      | Carter     |
| IA    | Herbert C.    | Hoover     |
| IL    | Barack        | Obama      |
| IL    | Ronald W.     | Reagan     |
| KY    | Abraham       | Lincoln    |
| MA    | George H.W.   | Bush       |
| MA    | John          | Adams      |
| MA    | John F.       | Kennedy    |
| MA    | John Quincy   | Adams      |
| MO    | Harry S       | Truman     |
| NC    | Andrew        | Johnson    |
| NC    | James K.      | Polk       |
| NE    | Gerald R.     | Ford       |
| . . . .                            |
+-------+---------------+------------+

```

ORDER BY Several Fields Introduces "Tuples"

Tuples provide crucial insight to structure of our records

- A 'tuple' is a subset of a record's fields
- For a given record, a tuple is that record's values for the specified fields
- Tuples are ordered -- (first_name, last_name) is a different tuple than (last_name, first_name)
- Note that records are, strictly speaking, tuples themselves, as they are ordered collections of fields
- At a 'query' level, a tuple is a specification of the fields we will look at for each particular record

We use tuples to specify unique or similar records, which helps us filter data

Data Structure / Records Structure

Analysis assumes some structure in the data

Data structure?

1. No patterns == Noise
2. Our records' values fall into patterns
3. Finding those patterns and interpreting them is the point of analysis

Tuples give us a path into structure of our data

We can use tuples to indicate unique records

If a particular set of fields has no more than 1 record for each combination of values in the table, that tuple indicates unique records

State and first_name fail for president -- we have several records with VA and James

How can we find unique records without visual inspection?

Getting Tuple Combinations

SELECT DISTINCT: tuple combinations

```
SELECT DISTINCT
    state,
    first_name
FROM
    sampdb.president
ORDER BY
    state,
    first_name

```

This returns the set of unique combinations of values for the tuple

No duplications

But can we tell if any have duplicates?

We need information about the records themselves -- a count of records by tuple field combination values

Just unique tuples

```
+-------+---------------+
| state | first_name    |
+-------+---------------+
| AR    | William J.    |
| CA    | Richard M.    |
| CT    | George W.     |
| GA    | James E.      |
| IA    | Herbert C.    |
| IL    | Barack        |
| IL    | Ronald W.     |
| KY    | Abraham       |
| MA    | George H.W.   |
| MA    | John          |
| MA    | John F.       |
| MA    | John Quincy   |
| MO    | Harry S       |
| NC    | Andrew        |
| NC    | James K.      |
| NE    | Gerald R.     |
| NH    | Franklin      |
| NJ    | Grover        |
| NY    | Franklin D.   |
| NY    | Martin        |
| NY    | Millard       |
| NY    | Theodore      |
| . . .                 |
+-------+---------------+

```

Aggregation

Aggregation Functions Describe Record Sets

COUNT(*) counts records

```
SELECT
    COUNT(*)
FROM
    sampdb.president
;

```

This returns the count of records returned by the query conditions

Here, the number of records in the table

Result

```
+----------+
| COUNT(*) |
+----------+
|       42 |
+----------+

```

COUNT()

COUNT()

- A "function" -- performs calculations based on arguments passed to it
- Previous query base, as an argument, the "all records" wild card
- An "aggregating" function -- it takes a set of records or values, and performs calculations on that set
- We will also see functions that act on values of only a single record
- COUNT() simply counts the (not NULL) values passed to it
- There is an exception for the * argument, COUNT() counts all records even if some have only NULL values

COUNT() Fields

COUNT([field name]) counts values in fields

```
SELECT
    COUNT(first_name),
    COUNT(death)
FROM
    sampdb.president
;

```

Result

```
+-------------------+--------------+
| COUNT(first_name) | COUNT(death) |
+-------------------+--------------+
|                42 |           38 |
+-------------------+--------------+

```

COUNT() Distinct Tuples

COUNT() distinct tuples

```
SELECT
    COUNT(DISTINCT state, first_name)
FROM
    sampdb.president

;

```

Result

```
+-----------------------------------+
| COUNT(DISTINCT state, first_name) |
+-----------------------------------+
|                                41 |
+-----------------------------------+

```

GROUP BY Subsets Data

GROUP BY divides records by unique tuples

Aggregation functions count within those subsets, not over the entire table

```
SELECT
    state,
    COUNT(*)
FROM
    sampdb.president
GROUP BY
    state
ORDER BY
    state
;

```

Result

```
+-------+----------+
| state | COUNT(*) |
+-------+----------+
| AR    |        1 |
| CA    |        1 |
| CT    |        1 |
| GA    |        1 |
| IA    |        1 |
| IL    |        1 |
| KY    |        1 |
| MA    |        4 |
| MO    |        1 |
| NC    |        2 |
| NE    |        1 |
| NH    |        1 |
| NJ    |        1 |
| NY    |        4 |
| OH    |        7 |
| PA    |        1 |
| SC    |        1 |
| TX    |        2 |
| VA    |        8 |
| VT    |        2 |
+-------+----------+

```

GROUP BY Tuple

We can GROUP BY tuples

```
SELECT
    state,
    first_name,
    COUNT(*)
FROM
    sampdb.president
GROUP BY
    state,
    first_name
ORDER BY
    COUNT(*) DESC,
    state,
    first_name
LIMIT
    10
;

```

So we can count records within tuples

Result

```
+-------+-------------+----------+
| state | first_name  | COUNT(*) |
+-------+-------------+----------+
| VA    | James       |        2 |
| AR    | William J.  |        1 |
| CA    | Richard M.  |        1 |
| CT    | George W.   |        1 |
| GA    | James E.    |        1 |
| IA    | Herbert C.  |        1 |
| IL    | Ronald W.   |        1 |
| KY    | Abraham     |        1 |
| MA    | George H.W. |        1 |
| MA    | John        |        1 |
+-------+-------------+----------+

```

Exciting, right?

Doesn't seem like much

- But notice that the syntax depends heavily on context
- Aggregating functions -- eg, COUNT() -- behave very differently based on the presence and detail of a GROUP BY clause
- Our queries define sets of records, return particular records or information about subsets of those records
- GROUP BY drives the collection of records into subsets for summary
- Doesn't seem like much, but provides a foundation for our next steps through the syntax

Practice

Write a query counting presidents by first_name

Order that query in descending order

Limit the output to the first five names

Do the same -- all three steps -- for last_name

Filtering -- WHERE and HAVING

Filtering the results

```
SELECT
    first_name,
    last_name,
    state
FROM
    sampdb.president
WHERE
    state = 'VA'
;

```

```
+------------+------------+-------+
| first_name | last_name  | state |
+------------+------------+-------+
| William H. | Harrison   | VA    |
| Thomas     | Jefferson  | VA    |
| James      | Madison    | VA    |
| James      | Monroe     | VA    |
| Zachary    | Taylor     | VA    |
| John       | Tyler      | VA    |
| George     | Washington | VA    |
| Woodrow    | Wilson     | VA    |
+------------+------------+-------+

```

- Notice the reorganization of the text for clarity

Filtering by comparison

```
SELECT
    first_name,
    last_name,
    birth,
    state
FROM
    sampdb.president
WHERE
    birth <= '1900-01-01'
ORDER BY
    birth
LIMIT 10
;
```

```
+-------------+------------+------------+-------+
| first_name  | last_name  | birth      | state |
+-------------+------------+------------+-------+
| George      | Washington | 1732-02-22 | VA    |
| John        | Adams      | 1735-10-30 | MA    |
| Thomas      | Jefferson  | 1743-04-13 | VA    |
| James       | Madison    | 1751-03-16 | VA    |
| James       | Monroe     | 1758-04-28 | VA    |
| Andrew      | Jackson    | 1767-03-15 | SC    |
| John Quincy | Adams      | 1767-07-11 | MA    |
| William H.  | Harrison   | 1773-02-09 | VA    |
| Martin      | Van Buren  | 1782-12-05 | NY    |
| Zachary     | Taylor     | 1784-11-24 | VA    |
+-------------+------------+------------+-------+
```

- Here we take only those presidents born before 20th century

Filtering affects set GROUPed

```
SELECT
    state,
    COUNT(*)
FROM
    sampdb.president
WHERE
    birth <= '1900-01-01'
GROUP BY
    state
ORDER BY
    COUNT(*)
;

```

```
+-------+----------+
| state | COUNT(*) |
+-------+----------+
| KY    |        1 |
| NH    |        1 |
| PA    |        1 |
| MO    |        1 |
| NJ    |        1 |
| TX    |        1 |
| IA    |        1 |
| SC    |        1 |
| MA    |        2 |
| VT    |        2 |
| NC    |        2 |
| NY    |        4 |
| OH    |        7 |
| VA    |        8 |
+-------+----------+

```

- We count here by state for presidents before 20th century

HAVING filters after GROUP

```
SELECT
    state,
    COUNT(*)
FROM
    sampdb.president
WHERE
    birth <= '1900-01-01'
GROUP BY
    state
HAVING
    COUNT(*) >= 2
ORDER BY
    COUNT(*)
;

```

```
+-------+----------+
| state | COUNT(*) |
+-------+----------+
| MA    |        2 |
| VT    |        2 |
| NC    |        2 |
| NY    |        4 |
| OH    |        7 |
| VA    |        8 |
+-------+----------+

```

- We count here by state for presidents before 20th century

Filter by list

```
SELECT
    first_name,
    last_name,
    state
FROM
    sampdb.president
WHERE
    state
        IN
    (
        'MA',
        'VA'
    )
ORDER BY
    state
;

```

```
+-------------+------------+-------+
| first_name  | last_name  | state |
+-------------+------------+-------+
| John        | Adams      | MA    |
| John Quincy | Adams      | MA    |
| George H.W. | Bush       | MA    |
| John F.     | Kennedy    | MA    |
| George      | Washington | VA    |
| John        | Tyler      | VA    |
| Zachary     | Taylor     | VA    |
| James       | Monroe     | VA    |
| James       | Madison    | VA    |
| Thomas      | Jefferson  | VA    |
| William H.  | Harrison   | VA    |
| Woodrow     | Wilson     | VA    |
+-------------+------------+-------+

```

- We use parentheses to define a list of state for which we want presidents

Filter by tuple

```
SELECT
    first_name,
    last_name,
    state
FROM
    sampdb.president
WHERE
    (last_name, state)
        IN
    (
        ('Adams', 'MA')
    )
;

```

```
+-------------+-----------+-------+
| first_name  | last_name | state |
+-------------+-----------+-------+
| John        | Adams     | MA    |
| John Quincy | Adams     | MA    |
+-------------+-----------+-------+

```

- And we can filter by tuple

Combining Conditions

```
SELECT
    first_name,
    last_name,
    state
FROM
    sampdb.president
WHERE
    state
        IN
    (
        'MA',
        'CA',
        'OH'
    )

    AND
        birth > '1900-01-01'
ORDER BY
    state
;

```

```
+-------------+-----------+-------+
| first_name  | last_name | state |
+-------------+-----------+-------+
| Richard M.  | Nixon     | CA    |
| George H.W. | Bush      | MA    |
| John F.     | Kennedy   | MA    |
+-------------+-----------+-------+

```

- We can combine conditions

Practice

Female students

Students with scores for test event 3

Students with scores for test event 3, with scores greater than 75

Presidents alive during the Civil War

Presidents alive during the Second World War

Record Level Functions

CONCAT() combines strings

```
SELECT
    first_name,
    last_name,
    CONCAT(first_name, ' ', last_name)
FROM
    sampdb.president
WHERE
    state = 'VA'
;

```

- CONCAT() combines two strings into one
- Unlike aggregation function (e.g. COUNT() et al), record functions operate only on the values of each particular record

```
+------------+------------+-----------------------------------+
| first_name | last_name  | CONCAT(first_name, ' ', last_name) |
+------------+------------+-----------------------------------+
| William H. | Harrison   | William H. Harrison               |
| Thomas     | Jefferson  | Thomas Jefferson                  |
| James      | Madison    | James Madison                     |
| James      | Monroe     | James Monroe                      |
| Zachary    | Taylor     | Zachary Taylor                    |
| John       | Tyler      | John Tyler                        |
| George     | Washington | George Washington                 |
| Woodrow    | Wilson     | Woodrow Wilson                    |
+------------+------------+-----------------------------------+

```

CONCAT() doesn't need _field_ values

```
SELECT
    last_name,
    state,
    CONCAT('Mickey', ' ', 'Mouse')
FROM
    sampdb.president
WHERE
    state = 'VA'
;

```

- Strings passed to CONCAT can be hard-coded values
- Note that the call on CONCAT() produces the same output for each record
- And, that there is one record output for each occurence record meeting the filter criterion

```
+------------+-------+--------------------------------+
| last_name  | state | CONCAT('Mickey', ' ', 'Mouse') |
+------------+-------+--------------------------------+
| Harrison   | VA    | Mickey Mouse                   |
| Jefferson  | VA    | Mickey Mouse                   |
| Madison    | VA    | Mickey Mouse                   |
| Monroe     | VA    | Mickey Mouse                   |
| Taylor     | VA    | Mickey Mouse                   |
| Tyler      | VA    | Mickey Mouse                   |
| Washington | VA    | Mickey Mouse                   |
| Wilson     | VA    | Mickey Mouse                   |
+------------+-------+--------------------------------+

```

We use AS to create 'alias' labels

```
SELECT
    first_name,
    last_name,
    CONCAT(first_name, ' ', last_name) AS full_name
FROM
    sampdb.president
WHERE
    state = 'VA'
;

```

- This gives a cleaner name to the field created by the function output

```
+------------+------------+---------------------+
| first_name | last_name  | full_name           |
+------------+------------+---------------------+
| William H. | Harrison   | William H. Harrison |
| Thomas     | Jefferson  | Thomas Jefferson    |
| James      | Madison    | James Madison       |
| James      | Monroe     | James Monroe        |
| Zachary    | Taylor     | Zachary Taylor      |
| John       | Tyler      | John Tyler          |
| George     | Washington | George Washington   |
| Woodrow    | Wilson     | Woodrow Wilson      |
+------------+------------+---------------------+
```

Field values can be simple terms in expressions

```
SELECT
    event_id, student_id,
    score,
    score * 2 AS doubled
FROM
    sampdb.score
LIMIT 10
;
```

- Here, for each record output, the value of the score field is simply doubled

```
+----------+------------+-------+---------+
| event_id | student_id | score | doubled |
+----------+------------+-------+---------+
|        1 |          1 |    20 |      40 |
|        1 |          3 |    20 |      40 |
|        1 |          4 |    18 |      36 |
|        1 |          5 |    13 |      26 |
|        1 |          6 |    18 |      36 |
|        1 |          7 |    14 |      28 |
|        1 |          8 |    14 |      28 |
|        1 |          9 |    11 |      22 |
|        1 |         10 |    19 |      38 |
|        1 |         11 |    18 |      36 |
+----------+------------+-------+---------+
```

DATE types permit date arithmetic

```
SELECT
    last_name,
    birth,
    death,
    DATEDIFF(death, birth) AS days,
    DATEDIFF(death, birth) / 365 AS years
FROM
    sampdb.president
WHERE
    death IS NOT NULL
LIMIT 10
;

```

- DATEDIFF() gives the number of days between two dates
- We can use its value as an expression term to roughly calculate age in years

```
+------------+------------+------------+-------+---------+
| last_name  | birth      | death      | days  | years   |
+------------+------------+------------+-------+---------+
| Adams      | 1735-10-30 | 1826-07-04 | 33119 | 90.7370 |
| Adams      | 1767-07-11 | 1848-02-23 | 29446 | 80.6740 |
| Arthur     | 1829-10-05 | 1886-11-18 | 20863 | 57.1589 |
| Buchanan   | 1791-04-23 | 1868-06-01 | 28163 | 77.1589 |
| Cleveland  | 1837-03-18 | 1908-06-24 | 26030 | 71.3151 |
| Coolidge   | 1872-07-04 | 1933-01-05 | 22099 | 60.5452 |
| Eisenhower | 1890-10-14 | 1969-03-28 | 28654 | 78.5041 |
| Fillmore   | 1800-01-07 | 1874-03-08 | 27088 | 74.2137 |
| Ford       | 1913-07-14 | 2006-12-26 | 34133 | 93.5151 |
| Garfield   | 1831-11-19 | 1881-09-19 | 18202 | 49.8685 |
+------------+------------+------------+-------+---------+

```

String functions: Return text and text positions

```
SELECT LOCATE(' ', 'cat on the mat') AS locate_example;

SELECT LEFT('cat on the mat', 3) AS left_example;

SELECT LEFT(
    'cat on the mat',
    LOCATE(' ', 'cat on the mat')
    ) AS nested_example;

SELECT MID('cat on the mat', 5, 2) AS mid_example;


```

```
+----------------+
| locate_example |
+----------------+
|              4 |
+----------------+
+--------------+
| left_example |
+--------------+
| cat          |
+--------------+
+----------------+
| nested_example |
+----------------+
| cat            |
+----------------+
+-------------+
| mid_example |
+-------------+
| on          |
+-------------+

```

Use string functions to parse text

```
SELECT
    first_name,
    LOCATE(' ', first_name) AS first_space,
    LEFT(first_name, LOCATE(' ', first_name) -1 ) AS just_first,
    LENGTH(first_name) AS string_length,
    mid(
        first_name,
        LOCATE(' ', first_name),
        LENGTH(first_name) - LOCATE(' ', first_name) + 1
    ) AS middle_name
FROM
    sampdb.president
;

```

Parsed output

```
+---------------+-------------+------------+---------------+-------------+
| first_name    | first_space | just_first | string_length | middle_name |
+---------------+-------------+------------+---------------+-------------+
| George        |           0 |            |             6 |             |
| John          |           0 |            |             4 |             |
| Thomas        |           0 |            |             6 |             |
| James         |           0 |            |             5 |             |
| James         |           0 |            |             5 |             |
| John Quincy   |           5 | John       |            11 |  Quincy     |
| Andrew        |           0 |            |             6 |             |
| Martin        |           0 |            |             6 |             |
| William H.    |           8 | William    |            10 |  H.         |
| John          |           0 |            |             4 |             |
| James K.      |           6 | James      |             8 |  K.         |
| Zachary       |           0 |            |             7 |             |
| . . .                                                              |
+---------------+-------------+------------+---------------+-------------+

```

Use Logic Functions to handle edge cases

```
SELECT
    first_name,
    LOCATE(' ', first_name) AS first_space,
    IF(
        LOCATE(' ', first_name) = 0,
        first_name,
        LEFT(first_name, LOCATE(' ', first_name) -1 )
    ) as proper_first,
    LEFT(first_name, LOCATE(' ', first_name) -1 ) AS just_first,
    mid(
        first_name,
        LOCATE(' ', first_name),
        LENGTH(first_name) - LOCATE(' ', first_name) + 1
    ) AS middle_name
FROM
    sampdb.president
;

```

Parsed output

first_name	first_space	proper_first	just_first	middle_name
George	0	George		
John	0	John		
Thomas	0	Thomas		
James	0	James		
James	0	James		
John Quincy	5	John	John	Quincy
Andrew	0	Andrew		
Martin	0	Martin		
William H.	8	William	William	H.
John	0	John		
James K.	6	James	James	K.
Zachary	0	Zachary		
. . .				

Practice Queries

Query returning just the first name of each president

Query counting presidents by month of birth. (The function expression MONTH(birth) will return the month of that field.)

Query returning just the 'proper' first name of each president

Query counting presidents by proper first name

The initials of each president

Query counting presidents by last initial

Functions and expressions can filter

```
SELECT
    last_name,
    birth,
    death,
    DATEDIFF(death, birth) AS days,
    DATEDIFF(death, birth) / 365 AS years
FROM
    sampdb.president
WHERE
        death IS NOT NULL
    -- fails:
    -- AND years < 70
    -- works:
    AND DATEDIFF(death, birth) / 365 < 70
LIMIT 10
;

```

- Notice that we can't use the alias 'years'
- We can use that same expression, though

```
+-----------+------------+------------+-------+---------+
| last_name | birth      | death      | days  | years   |
+-----------+------------+------------+-------+---------+
| Arthur    | 1829-10-05 | 1886-11-18 | 20863 | 57.1589 |
| Coolidge  | 1872-07-04 | 1933-01-05 | 22099 | 60.5452 |
```

Logical Functions

CASE is the SQL standard

This works in both MySQL and Postgres

```
SELECT
    CASE WHEN 2 > 1
        THEN 'it is true'
        ELSE 'it is false'
    END AS case_example
;

```

```
+--------------+
| case_example |
+--------------+
| it is true   |
+--------------+

```

MySQL provides a more familiar IF()

This won't work in Postgres

You could define a Postgres function for it

```
SELECT
    IF( 2 > 1,
        'It is true',
        'It is false'
    ) AS if_example
;
-- IF() function not available in postgres?

```

```
+------------+
| if_example |
+------------+
| It is true |
+------------+

```

CASE statements allow multiple conditions

We needn't nest CASE, we can add conditions

```
SELECT
    CASE
        WHEN 10 <= 5
            THEN 'less than five'
        WHEN 10 > 5 AND 10 < 8
            THEN 'between five and eight'
        WHEN 10 < 10
            THEN 'between eight and ten'
        ELSE 'it is ten or greater'
    END AS case_example
;

```

```
+----------------------+
| case_example         |
+----------------------+
| it is ten or greater |
+----------------------+

```

Combining Tables -- JOIN

Finding Megan's scores

Find Megan's student_id

```
SELECT
        name
    ,   student_id
FROM
    sampdb.student
WHERE
    name = 'Megan'
;

```

```
+-------+------------+
| name  | student_id |
+-------+------------+
| Megan |          1 |
+-------+------------+

```

Get the scores for that student_id

```
SELECT
        student_id
    ,   event_id
    ,   score

FROM
    sampdb.score

WHERE
    student_id = 1
;

```

```
+------------+----------+-------+
| student_id | event_id | score |
+------------+----------+-------+
|          1 |        1 |    20 |
|          1 |        2 |    17 |
|          1 |        3 |    88 |
|          1 |        5 |    15 |
|          1 |        6 |   100 |
+------------+----------+-------+

```

But we would rather have the name on that table

```
SELECT
        sampdb.student.name
    ,   sampdb.student.student_id
    ,   sampdb.score.event_id
    ,   sampdb.score.score
FROM
    sampdb.student
INNER JOIN
    sampdb.score
ON
    sampdb.student.student_id =
        sampdb.score.student_id
WHERE
    sampdb.student.name = 'Megan'
;

```

```
+-------+------------+----------+-------+
| name  | student_id | event_id | score |
+-------+------------+----------+-------+
| Megan |          1 |        1 |    20 |
| Megan |          1 |        2 |    17 |
| Megan |          1 |        3 |    88 |
| Megan |          1 |        5 |    15 |
| Megan |          1 |        6 |   100 |
+-------+------------+----------+-------+

```

Joins

Two Really Simple Tables

```
+------+------+
| i1   | c1   |
+------+------+
|    1 | a    |
|    2 | b    |
|    3 | c    |
+------+------+

```

```
+------+------+
| i2   | c2   |
+------+------+
|    2 | c    |
|    3 | b    |
|    4 | a    |
+------+------+

```

Inner Join

```
SELECT * FROM join_sample.t1 INNER JOIN join_sample.t2;

```

```
+------+------+------+------+
| i1   | c1   | i2   | c2   |
+------+------+------+------+
|    1 | a    |    2 | c    |
|    2 | b    |    2 | c    |
|    3 | c    |    2 | c    |
|    1 | a    |    3 | b    |
|    2 | b    |    3 | b    |
|    3 | c    |    3 | b    |
|    1 | a    |    4 | a    |
|    2 | b    |    4 | a    |
|    3 | c    |    4 | a    |
+------+------+------+------+

```

Limiting Inner Join Results

```
SELECT
    *
FROM
    join_sample.t1
INNER JOIN
    join_sample.t2
WHERE
    join_sample.t1.i1 = join_sample.t2.i2
;

```

Left Join

Get all rows from the "left" table, each with that row from the "right" table that matches on the specified fields

```
SELECT
    *
FROM
    join_sample.t1
LEFT JOIN
    join_sample.t2
    ON
        join_sample.t1.i1 = join_sample.t2.i2
;

```

```
+------+------+------+------+
| i1   | c1   | i2   | c2   |
+------+------+------+------+
|    1 | a    | NULL | NULL |
|    2 | b    |    2 | c    |
|    3 | c    |    3 | b    |
+------+------+------+------+

```

Breaking down the query

A join combines rows from different tables

The "ON" clause specifies conditions for combining records

Here, records with the same student_id value are combined into a single record

The SELECT clause still specifies which fields are displayed from that combined record

INNER JOIN specifies which table to join data from

The addition of "scr" and "st" in the specification of tables provides alias values for reference by the other clauses

Without these the query wouldn't know which table it should find a field in

```
SELECT
    st.name AS          Name,
    scr.student_id AS   Id,
    AVG(scr.score) AS   Average,
    COUNT(scr.score) AS "# Tests"
FROM
    sampdb.score scr
INNER JOIN
    sampdb.student st
    ON
        scr.student_id = st.student_id
GROUP BY
    scr.student_id
ORDER BY
    Average DESC
LIMIT 5
;

```

Practice

Join all rows in sampdb.student with all rows in sampdb.score

As above, but filter to match each score to the proper student

Output employee name and their sales

More on JOINS

Data is spread over several tables

Title Data

```
+------------+--------------+------+-----+---------+-------+
| Field      | Type         | Null | Key | Default | Extra |
+------------+--------------+------+-----+---------+-------+
| title_id   | char(3)      | NO   | PRI | NULL    |       |
| title_name | varchar(40)  | NO   |     | NULL    |       |
| type       | varchar(10)  | YES  |     | NULL    |       |
| pub_id     | char(3)      | NO   |     | NULL    |       |
| pages      | int(11)      | YES  |     | NULL    |       |
| price      | decimal(5,2) | YES  |     | NULL    |       |
| sales      | int(11)      | YES  |     | NULL    |       |
| pubdate    | date         | YES  |     | NULL    |       |
| contract   | smallint(6)  | NO   |     | NULL    |       |
+------------+--------------+------+-----+---------+-------+
```

Title and Author do NOT join directly

Authors

```
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| au_id    | char(3)     | NO   | PRI | NULL    |       |
| au_fname | varchar(15) | NO   |     | NULL    |       |
| au_lname | varchar(15) | NO   |     | NULL    |       |
| phone    | varchar(12) | YES  |     | NULL    |       |
| address  | varchar(20) | YES  |     | NULL    |       |
| city     | varchar(15) | YES  |     | NULL    |       |
| state    | char(2)     | YES  |     | NULL    |       |
| zip      | char(5)     | YES  |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

Title and Authors

```
+---------------+--------------+------+-----+---------+-------+
| Field         | Type         | Null | Key | Default | Extra |
+---------------+--------------+------+-----+---------+-------+
| title_id      | char(3)      | NO   | PRI | NULL    |       |
| au_id         | char(3)      | NO   | PRI | NULL    |       |
| au_order      | smallint(6)  | NO   |     | NULL    |       |
| royalty_share | decimal(5,2) | NO   |     | NULL    |       |
+---------------+--------------+------+-----+---------+-------+
```

- Notice that this table serves to connect authors to their titles

Getting books written

```
-- List books each author wrote
-- Fehily 7.7

SELECT
    a.au_id,
    a.au_fname,
    a.au_lname,
    ta.title_id,
    ta.au_id AS ta_author_id
FROM
    books.authors a
INNER JOIN
    books.title_authors ta
    ON a.au_id = ta.au_id
ORDER BY
    a.au_id ASC,
    ta.title_id ASC
;

```

```
+-------+-----------+-----------+----------+--------------+
| au_id | au_fname  | au_lname  | title_id | ta_author_id |
+-------+-----------+-----------+----------+--------------+
| A01   | Sarah     | Buchman   | T01      | A01          |
| A01   | Sarah     | Buchman   | T02      | A01          |
| A01   | Sarah     | Buchman   | T13      | A01          |
| A02   | Wendy     | Heydemark | T06      | A02          |
| A02   | Wendy     | Heydemark | T07      | A02          |
| A02   | Wendy     | Heydemark | T10      | A02          |
| A02   | Wendy     | Heydemark | T12      | A02          |
| A03   | Hallie    | Hull      | T04      | A03          |
| A03   | Hallie    | Hull      | T11      | A03          |
| A04   | Klee      | Hull      | T04      | A04          |
| A04   | Klee      | Hull      | T05      | A04          |
| A04   | Klee      | Hull      | T07      | A04          |
| A04   | Klee      | Hull      | T11      | A04          |
| A05   | Christian | Kells     | T03      | A05          |
| A06   |           | Kellsey   | T08      | A06          |
| A06   |           | Kellsey   | T09      | A06          |
| A06   |           | Kellsey   | T11      | A06          |
+-------+-----------+-----------+----------+--------------+

```

Getting books titles

```
SELECT
    t.title_id,
    t.title_name,
    ta.title_id
FROM
    books.titles t
INNER JOIN
    books.title_authors ta
ON
    t.title_id = ta.title_id
;

```

```
+----------+-----------------------------------+----------+
| title_id | title_name                        | title_id |
+----------+-----------------------------------+----------+
| T01      | 1977!                             | T01      |
| T02      | 200 Years of German Humor         | T02      |
| T03      | Ask Your System Administrator     | T03      |
| T04      | But I Did It Unconsciously        | T04      |
| T04      | But I Did It Unconsciously        | T04      |
| T05      | Exchange of Platitudes            | T05      |
| T06      | How About Never?                  | T06      |
| T07      | I Blame My Mother                 | T07      |
| T07      | I Blame My Mother                 | T07      |
| T08      | Just Wait Until After School      | T08      |
| T09      | Kiss My Boo-Boo                   | T09      |
| T10      | Not Without My Faberge Egg        | T10      |
| T11      | Perhaps It's a Glandular Problem  | T11      |
| T11      | Perhaps It's a Glandular Problem  | T11      |
| T11      | Perhaps It's a Glandular Problem  | T11      |
| T12      | Spontaneous, Not Annoying         | T12      |
| T13      | What Are The Civilian Applications? | T13      |
+----------+-----------------------------------+----------+

```

A multi-table JOIN connects these in one query

Two JOINs in one query

```
SELECT
    ttl.title_name,
    -- ttl.title_id,
    -- ta.au_id,
    auth.au_fname,
    auth.au_lname

FROM
    books.titles AS ttl
INNER JOIN
    books.title_authors AS ta
INNER JOIN
    books.authors AS auth
WHERE
    ttl.title_id = ta.title_id
    AND ta.au_id = auth.au_id
;


```

- Remember, INNER JOIN makes a 'super table'
- Here, this table combines three base tables
- Of that table, rows are returns on matching title and author ids
- Note that no fields from title_authors are displayed
- title_authors here serves to 'bridge' the one table to another
- This structure allows the data model to handle authors with multiple titles, and titles with multiple authors

Title and Author in one table

```
SELECT
    ttl.title_name,
    auth.au_fname,
    auth.au_lname

FROM
    books.titles AS ttl

INNER JOIN
    books.title_authors AS ta
ON
 ttl.title_id = ta.title_id

INNER JOIN
    books.authors AS auth
ON
 ta.au_id = auth.au_id

;


```

Same query, using ON rather than WHERE

```
SELECT
    ttl.title_name,
    auth.au_fname,
    auth.au_lname

FROM
    books.titles AS ttl

INNER JOIN
    books.title_authors AS ta
ON
 ttl.title_id = ta.title_id

INNER JOIN
    books.authors AS auth
ON
 ta.au_id = auth.au_id

;


```

We can join a table to itself

Accesses a table's data in reference to itself

```
SELECT
    au_1.au_id,
    au_1.au_fname,
    au_1.au_lname,
    au_1.state
FROM
    books.authors au_1
INNER JOIN
    books.authors au_2
ON
    au_1.state = au_2.state
WHERE
    au_2.au_id = 'A04'

;

```

```
+-------+----------+----------+-------+
| au_id | au_fname | au_lname | state |
+-------+----------+----------+-------+
| A03   | Hallie   | Hull     | CA    |
| A04   | Klee     | Hull     | CA    |
| A06   |          | Kellsey  | CA    |
+-------+----------+----------+-------+

```

Self joins can enable hierarchy extraction

Consider a table of employee / boss data

```
+--------+------------------+---------+
| emp_id | emp_name         | boss_id |
+--------+------------------+---------+
| E01    | Lord Copper      | NULL    |
| E02    | Jocelyn Hitchcock| E01     |
| E03    | Mr. Salter       | E01     |
| E04    | William Boot     | E03     |
| E05    | Mr. Corker       | E03     |
+--------+------------------+---------+

```

- The table is self-referential, in that each boss_id value is found in employee_id
- We can use a self-join to get a table of who reports to whom

Hierarchy extraction (cont)

```
SELECT
    e1.emp_name AS "Employee Name",
    e2.emp_name AS "Boss Name",
    e1.boss_id,
    e2.emp_id
FROM
    books.employees e1
INNER JOIN
    books.employees e2
ON
    e1.boss_id = e2.emp_id
;

```

```
+-------------------+-------------+
| Employee Name     | Boss Name   |
+-------------------+-------------+
| Jocelyn Hitchcock | Lord Copper |
| Mr. Salter        | Lord Copper |
| William Boot      | Mr. Salter  |
| Mr. Corker        | Mr. Salter  |
+-------------------+-------------+

```

Here, the ON condition selects those rows that match an employee with their boss

Practice Problems

From books, employee name with employee sales

Subqueries -- Referring to SQL results

Subqueries embed one query within another

These embedded queries are 'subqueries'

We have to watch the table returned by the subquery carefully

The returned table has to have the fields and row expected of it by the calling query

Two varieties, 'correlated' and 'uncorrelated'

'correlated' is cooler but harder, we'll do 'uncorrelated' first

Get names of publishers of biography

Step one: Get pub_id of the publishers

```
SELECT
    pub_id
FROM
    books.titles
WHERE
    type = 'biography'
;
```

```
+--------+
| pub_id |
+--------+
| P01    |
| P03    |
| P01    |
| P01    |
+--------+
```

Get names of publishers of biography -- cont.

Step two: Get names of those publishers

```
SELECT
    pub_name
FROM
    books.publishers
WHERE
    pub_id IN ('P01', 'P03')
;
```

```
+----------------------+
| pub_name             |
+----------------------+
| Abatis Publishers    |
| Schadenfreude Press  |
+----------------------+
```

Get names of publishers of biography -- cont.

But we can do this in one query

```
SELECT
    pub_name
FROM
    books.publishers
WHERE
    pub_id IN (
        SELECT
            pub_id
        FROM
            books.titles
        WHERE
            type = 'biography'
    )
;
```

```
+----------------------+
| pub_name             |
+----------------------+
| Abatis Publishers    |
| Schadenfreude Press  |
+----------------------+
```

Another example

Get authors who have not written a book

```
SELECT
    au_id, au_fname, au_lname
FROM
    books.authors
WHERE
    au_id NOT IN
    (SELECT au_id FROM books.title_authors)
;

```

```
+-------+----------+-------------+
| au_id | au_fname | au_lname    |
+-------+----------+-------------+
| A07   | Paddy    | O'Furniture |
+-------+----------+-------------+

```

More complex example

Say we want a list of students scoring above on grade event 3

Getting the score is easy:

```
SELECT
    event_id,
    AVG(score)
FROM
    sampdb.score
WHERE
        event_id = 3
GROUP BY
    event_id
LIMIT 5
;

```

Filtering is easy:

```
SELECT
    student_id AS student,
    event_id AS event,
    score
FROM
    sampdb.score
WHERE
        event_id = 3
    AND score > 78.2258
;

```

Results

Results

```
+---------+-------+-------+
| student | event | score |
+---------+-------+-------+
|       1 |     3 |    88 |
|       2 |     3 |    84 |
|       5 |     3 |    97 |
|       6 |     3 |    83 |
|       7 |     3 |    88 |
+---------+-------+-------+
```

But:

What if the data change?

We want results for another event?

Better: refer directly to the average

```
SELECT
    student_id,
    event_id,
    score
FROM
    sampdb.score
WHERE
        event_id = 3
    AND score > (
        SELECT
            AVG(score)
        FROM
            sampdb.score
        WHERE
            event_id = 3
        GROUP BY
            event_id
    )
LIMIT 5;

```

```
+------------+-------+
| student_id | score |
+------------+-------+
|          1 |    88 |
|          2 |    84 |
|          5 |    97 |
|          6 |    83 |
|          7 |    88 |
+------------+-------+

```

Now:

- query automatically changes output for any data change
- we can get a similar list simply by changing the event_id we are seeking
- BUT -- what if we change event_id in one place, and not the other . . . ?

Better still: Remove duplication

```
SELECT
  student_id,
  score
FROM
  sampdb.score AS scr
WHERE
  event_id = 3
  AND score > (
    SELECT
      AVG(score)
    FROM
      sampdb.score
    WHERE
      event_id=scr.event_id
    GROUP BY
      event_id
  )
LIMIT 5;

```

```
+------------+-------+
| student_id | score |
+------------+-------+
|          1 |    88 |
|          2 |    84 |
|          5 |    97 |
|          6 |    83 |
|          7 |    88 |
+------------+-------+

```

Now:

- The subquery's event_id filter looks to each particular record for the value of event_id used in its filter
- Thus our query only refers to the desired event_id once
- e.g., we cannot make the 'event_id values out of step' mistake
- This is a 'correlated subquery', and we will spend more time on them later

Correlated queries send values from outer query to inner

```
-- List the books that have sales greater than or
-- equal to the average sales for books of its type
-- fehily 8.13

SELECT
    candidate.title_id,
    candidate.type,
    candidate.sales,
    (SELECT ROUND(AVG(sales), 1)
     FROM books.titles average
     WHERE average.type = candidate.type) AS type_average
FROM
    books.titles candidate
WHERE
    sales >=
    (SELECT AVG(sales)
     FROM books.titles average
     WHERE average.type = candidate.type)
;

```

```
+----------+-------------+---------+--------------+
| title_id | type        | sales   | type_average |
+----------+-------------+---------+--------------+
| T02      | history     |    9566 |       6866.3 |
| T03      | computer    |   25667 |       25667.0 |
| T05      | psychology  |  201440 |      102854.7 |
| T07      | biography   | 1500200 |      537173.7 |
| T09      | children    |    5000 |       4547.5 |
| T13      | history     |   10467 |       6866.3 |
+----------+-------------+---------+--------------+

```

Each score event has its own average and std. dev

```
SELECT
    event_id,
    AVG(score) AS average,
    STD(score) AS standard_deviation
FROM
    sampdb.score
GROUP BY
    event_id
;

```

```
+----------+---------+--------------------+
| event_id | average | standard_deviation |
+----------+---------+--------------------+
|        1 | 15.1379 |             3.7849 |
|        2 | 14.1667 |             3.7424 |
|        3 | 78.2258 |             9.2063 |
|        4 | 14.0370 |             3.8344 |
|        5 | 14.1852 |             3.3448 |
|        6 | 80.1724 |            12.4404 |
+----------+---------+--------------------+

```

Each student_id in score has multiple events

```
SELECT
    student_id,
    COUNT(event_id) as events
FROM
    sampdb.score
GROUP BY
    student_id
LIMIT 10;

```

```
+------------+--------+
| student_id | events |
+------------+--------+
|          1 |      5 |
|          2 |      5 |
|          3 |      6 |
|          4 |      5 |
|          5 |      6 |
|          6 |      6 |
|          7 |      6 |
|          8 |      6 |
|          9 |      6 |
|         10 |      6 |
+------------+--------+

```

We can find statistics for each of those events

```
SELECT
    student_id,
    event_id,
    score,
    (  SELECT ROUND(AVG(score), 1)
        FROM sampdb.score AS s2
        WHERE s1.event_id = s2.event_id
        GROUP BY event_id
    ) AS e_avg,
    (  SELECT ROUND(STD(score), 1)
        FROM sampdb.score AS s2
        WHERE s1.event_id = s2.event_id
        GROUP BY event_id
    ) AS e_dev
FROM
    sampdb.score AS s1
WHERE
    student_id = 1
;
```

```
+------------+----------+-------+-------+-------+
| student_id | event_id | score | e_avg | e_dev |
+------------+----------+-------+-------+-------+
|          1 |        1 |    20 |  15.1 |   3.8 |
|          1 |        2 |    17 |  14.2 |   3.7 |
|          1 |        3 |    88 |  78.2 |   9.2 |
|          1 |        5 |    15 |  14.2 |   3.3 |
|          1 |        6 |   100 |  80.2 |  12.4 |
+------------+----------+-------+-------+-------+
```

Get those statistics, for every event, for every record

```
SELECT
    score, student_id AS stdnt, event_id AS vnt,
    event_average AS vnt_avg, std_dev AS std,
    ( score - event_average ) / std_dev AS z_score
FROM
(
    SELECT
        s1.score, s1.student_id, s1.event_id,
        (  SELECT AVG(score)
            FROM sampdb.score AS s2
            WHERE s2.event_id = s1.event_id
            GROUP BY event_id
        ) AS event_average,
        (   SELECT STD(score)
            FROM sampdb.score AS s2
            WHERE s2.event_id = s1.event_id
            GROUP BY event_id
        ) AS std_dev
    FROM sampdb.score AS s1
) AS tmp

LIMIT 5
;

```

Result

```
+-------+-------+-----+---------+--------+-------------+
| score | stdnt | vnt | vnt_avg | std    | z_score     |
+-------+-------+-----+---------+--------+-------------+
|    20 |     1 |   1 | 15.1379 | 3.7849 |  1.28458961 |
|    20 |     3 |   1 | 15.1379 | 3.7849 |  1.28458961 |
|    18 |     4 |   1 | 15.1379 | 3.7849 |  0.75618024 |
|    13 |     5 |   1 | 15.1379 | 3.7849 | -0.56484320 |
|    18 |     6 |   1 | 15.1379 | 3.7849 |  0.75618024 |
+-------+-------+-----+---------+--------+-------------+
```

Correlated Subqueries work record by record

The 'inner' queries are run for each record in the 'outer' query

Definitions

- 'Outer' query is the query calling on result
- 'Inner' query supplies result
- Records in separate are referenced by aliases

Outer query checks inner query for each of its own records

1. Outer gets record
2. Outer consults inner, 'correlated' query
3. Inner query takes values from record in process for outer query
4. Inner passes result out to outer query

Are duplicative inner queries cached?

- It depends
- Performance is complex, requires first mastering syntax

A convoluted problem:

Say we want a list of those presidents from the three states that have sent the most presidents

We can get the list by a query:

```
SELECT
    state,
    COUNT(*) AS state_count
FROM
    sampdb.president
GROUP BY
    state
ORDER BY
    state_count DESC
LIMIT 3
;

```

```
+-------+-------------+
| state | state_count |
+-------+-------------+
| VA    |           8 |
| OH    |           7 |
| MA    |           4 |
+-------+-------------+

```

First, get criteria right

Revise our condition for state inclusion

```
SELECT
  MIN(st_count)
FROM
(
  SELECT
    state,
    COUNT(*) AS st_count
  FROM
    sampdb.president
  GROUP BY
    state
  ORDER BY
    state_count DESC
  LIMIT 3
) AS tmp
;

```

```
+------------------+
| MIN(st_count) |
+------------------+
|                4 |
+------------------+

```

Revise listing query

Uses presidents count criteria for state inclusion

```
SELECT
  state, COUNT(*) as state_count
FROM
  sampdb.president
GROUP BY
  state
HAVING
  COUNT(*) >=
  (
    SELECT
      MIN(state_count)
    FROM
      (
        SELECT
          state,
          COUNT(*) AS state_count
        FROM
          sampdb.president
        GROUP BY
          state
        ORDER BY
          state_count DESC
        LIMIT 3
      ) AS t3
  )
ORDER BY state_count DESC
;

```

```
+-------+-------------+
| state | state_count |
+-------+-------------+
| VA    |           8 |
| OH    |           7 |
| MA    |           4 |
| NY    |           4 |
+-------+-------------+

```

- This gives us a 'dynamic' derivation of the state list we need
- HAVING
- - similar to WHERE
- - appears after GROUP BY
- - refers to, and filters on, the grouped records and aggregated fields
- So WHERE filters the records that are aggregated into groups, and HAVING filters the groups themselves

Putting all this together provides our answer

```
SELECT last_name, state
FROM sampdb.president
WHERE
  state IN
  (
    SELECT state
    FROM sampdb.president
    GROUP BY state
    HAVING
      COUNT(*) >=
      (
        SELECT
          MIN(state_count)
        FROM
          (
            SELECT COUNT(*) AS state_count
            FROM sampdb.president
            GROUP BY state
            ORDER BY state_count DESC
            LIMIT 3
          ) AS t3
      )
  )
ORDER BY
  state, last_name, first_name
;
```

```
+------------+-------+
| last_name  | state |
+------------+-------+
| Adams      | MA    |
| Adams      | MA    |
| Bush       | MA    |
| Kennedy    | MA    |
| Fillmore   | NY    |
| Roosevelt  | NY    |
| Roosevelt  | NY    |
| Van Buren  | NY    |
| Garfield   | OH    |
| Grant      | OH    |
| Harding    | OH    |
| Harrison   | OH    |
| Hayes      | OH    |
| McKinley   | OH    |
| Taft       | OH    |
| Harrison   | VA    |
| Jefferson  | VA    |
| Madison    | VA    |
| Monroe     | VA    |
| Taylor     | VA    |
| Tyler      | VA    |
| Washington | VA    |
| Wilson     | VA    |
+------------+-------+
```

- Inner query lists the top three states and their counts
- Next extracts the minimum value of those counts
- Next out lists the states with that count or higher
- Last query takes presidents whose states fall in that list

Subqueries in SELECT field expressions

```
SELECT
( SELECT
    ROUND(AVG(score),1)
  FROM sampdb.score
  WHERE event_id = 1
  GROUP BY event_id
)   AS 1_scr,
( SELECT
    ROUND(AVG(score),1)
  FROM sampdb.score
  WHERE event_id = 2
  GROUP BY event_id
)   AS 2_scr,
( SELECT
    ROUND(AVG(score),1)
  FROM sampdb.score
  WHERE event_id = 3
  GROUP BY event_id
)   AS 3_scr

-- Notice the SELECT
-- expression is
-- just a field list

;
```

```
+-------+-------+-------+
| 1_scr | 2_scr | 3_scr |
+-------+-------+-------+
|  15.1 |  14.2 |  78.2 |
+-------+-------+-------+
```

Subqueries can be used in field expressions

- Here we use them to replicate an Excel pivot table
- Notice that this doesn't handle "arbitrary" columns
- We have to designate a field in our output for each column
- The fundamental problem is we can specify sets of records, but not fields
- There are hacks around this, but they're complicated

HAVING

Seen earlier in subquery

HAVING applies criteria to output of aggregations

- Calculated after WHERE and GROUP BY
- WHERE filters input records to initial query
- HAVING filters resultant aggregations

HAVING Example

States with more than one President whose last name doesn't begin with a vowel

```
SELECT
  state,
  COUNT(*) AS state_count
FROM
  sampdb.president
WHERE
  LEFT(last_name, 1)
  NOT IN
  ('A', 'E', 'I', '
    O', 'U')
GROUP BY
  state
HAVING
  state_count >= 2
;


```

```
+-------+-------------+
| state | state_count |
+-------+-------------+
| MA    |           2 |
| NC    |           2 |
| NY    |           4 |
| OH    |           7 |
| VA    |           8 |
+-------+-------------+

```

- WHERE clause strips out the vowel last names
- HAVING gets the states with more than one (now not-vowel) president
- Splitting up the WHERE and HAVING criteria gives a finer control over the output set

Practice

Revise query to get below average scores for event 2

Employees with above average sales

Presidents who lived longer than average

The age of the third longest lived President

The name of the third longest lived President

Queries for Analysis

Using queries for analysis

1. **Notice that all of our queries return (e.g., output) tables**

   The output is a set of records, with defined fields

2. **We can perform further operations on those returned tables, and join them to one another**

3. **By cascading queries, we can filter and massage our data to get where we want**

   "views" give us a tool for holding queries "in place"

Create a "base table"

- "Stores" the result of the SELECT query in a "table" available to other queries
- Uses functions to create an "age" column from birth and death

```
DROP VIEW IF EXISTS sampdb.president_age;
CREATE VIEW
    sampdb.president_age
AS
    SELECT
        last_name,
        state,
        ROUND(DATEDIFF(death, birth) / 365, 1) as AGE
    FROM
        sampdb.president ;

SELECT * FROM sampdb.president_age LIMIT 5;


```

```
+------------+-------+------+
| last_name  | state | AGE  |
+------------+-------+------+
| Washington | VA    | 67.9 |
| Adams      | MA    | 90.7 |
| Jefferson  | VA    | 83.3 |
| Madison    | VA    | 85.3 |
| Monroe     | VA    | 73.2 |
+------------+-------+------+

```

Derive a second table from our base

- Uses our prior query to create a table of aggregated values

```
DROP VIEW IF EXISTS sampdb.president_aggregates;
CREATE VIEW
    sampdb.president_aggregates
AS
    SELECT
        state,
        COUNT(state) as StateCount,
        AVG(age) as AverageAge,
        MAX(age) as MaxAge,
        MIN(age) as MinAge
    FROM
        sampdb.president_age
    GROUP BY
        state
    ORDER BY
        StateCount DESC
;
SELECT * FROM sampdb.president_aggregates LIMIT 5;


```

```
+-------+------------+------------+--------+--------+
| state | StateCount | AverageAge | MaxAge | MinAge |
+-------+------------+------------+--------+--------+
| VA    |          8 |   72.83750 |   85.3 |   65.7 |
| OH    |          7 |   62.87143 |   72.5 |   49.9 |
| MA    |          4 |   72.63333 |   90.7 |   46.5 |
| NY    |          4 |   69.32500 |   79.7 |   60.2 |
| TX    |          2 |   71.45000 |   78.5 |   64.4 |
+-------+------------+------------+--------+--------+

```

Combine the two to get our result

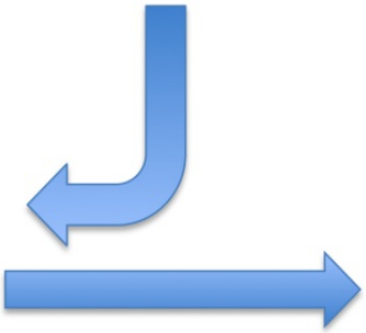
- Now we can join our Age table with our Aggregates table to get oldest presidents by State

```
SELECT
    sub.state,
    sub.last_name,
    sub.age
FROM
    sampdb.president_age sub
INNER JOIN
    sampdb.president_aggregates sub2
ON
    sub.state = sub2.state
    AND sub.age = sub2.MaxAge
LIMIT 5
;


```

```
+-------+-----------+------+
| state | last_name | AGE  |
+-------+-----------+------+
| MA    | Adams     | 90.7 |
| VA    | Madison   | 85.3 |
| SC    | Jackson   | 78.3 |
| NY    | Van Buren | 79.7 |
| NH    | Pierce    | 64.9 |
+-------+-----------+------+

```

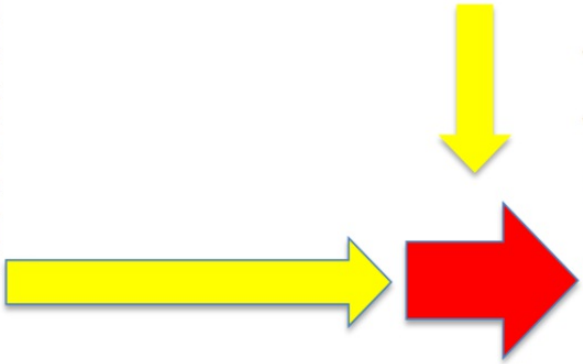
Cascading through queries to result

last_name	first_name	suffix	city	state	birth	death
Washington	George	NULL	Wakefield	VA	1732-02-22	1799-12-14
Adams	John	NULL	Braintree	MA	1735-10-30	1826-07-04
Jefferson	Thomas	NULL	Albemarle County	VA	1743-04-13	1826-07-04
Madison	James	NULL	Port Conway	VA	1751-03-16	1836-06-28
Monroe	James	NULL	Westmoreland County	VA	1758-04-28	1831-07-04
Adams	John Quincy	NULL	Braintree	MA	1767-07-11	1848-02-23



last_name	state	age
Washington	VA	67.9
Adams	MA	90.7
Jefferson	VA	83.3
Madison	VA	85.3
Monroe	VA	73.2
Adams	MA	80.7
Jackson	SC	78.3
Van Buren	NY	79.7
Harrison	VA	68.2
Tyler	VA	71.9
Polk	NC	53.7
Taylor	VA	65.7
Fillmore	NY	74.2

state	StateCount	AverageAge	MaxAge	MinAge
VA	8	72.83750	85.3	65.7
OH	7	62.87143	72.5	49.9
MA	4	72.63333	90.7	46.5
NY	4	69.32500	79.7	60.2
TX	2	71.45000	78.5	64.4
NC	2	60.15000	66.6	53.7



state	last_name	age
MA	Adams	90.7
VA	Madison	85.3
SC	Jackson	78.3
NY	Van Buren	79.7
NH	Pierce	64.9
PA	Buchanan	77.2

Can we do without the views?

- Here we use a subquery
- We're still combining tables with one another, but here we generate one of them on the fly

```
#   Gets oldest president, by state, without intervening views
#   -   better than only matching age to max age values b/c it avoids the
#       possibility of duplicate max age values

SELECT
    state,
    last_name AS oldest,
    ROUND(DATEDIFF(death, birth) / 365, 1) AS age

FROM
    sampdb.president

WHERE
   (    state,
        ROUND(DATEDIFF(death, birth) / 365, 1)
    ) IN

    (
        SELECT
            state,
            MAX(ROUND(DATEDIFF(death, birth) / 365, 1)) AS age
        FROM
            sampdb.president
        GROUP BY
            state
    )
;


```

Loading Data

Annoying but necessary

1. **No fundamental concepts involved here - just file management**
2. **Understanding this process does orient us to the fundamentals of what the computer is doing**
3. **We will go over this fast and revisit in exercises**

   Next week we'll go over problems in those

First, get data to database

1. ## Get data into CSV format

   Excel can do this for us

   On Mac:

2. ## "Comma Separated Values" will have CR line terminators

3. ## "Windows CSV" will have CRLF terminators

4. ## Line terminators? Hang on

Create a database

Simple query creating a home for our table

```
DROP DATABASE IF EXISTS learning;
CREATE DATABASE learning;

```

Create a table

- Very simple table

Notice homogeneity of data type (all varchar)

```
DROP TABLE IF EXISTS learning.from_excel;
CREATE TABLE learning.from_excel (

    date varchar(20),
    category varchar(20),
    event_id varchar(20)
);

```

Load data

Let's break this down

```
#   Loads a CSV file, saved from Excel with "Comma Separated Values" option
#   -   not "Windows" csv, not "MS-DOS" csv
#   -   Excel 2008 for Mac, 12.3.5

#   file command output: ASCII text, with CR line terminators

LOAD DATA LOCAL

    #   EDIT THIS PATH:
    INFILE "/Users/gimli/Desktop/from_excel.csv"

    INTO TABLE learning.from_excel

    FIELDS
        TERMINATED BY ','
        ENCLOSED BY '"'

    LINES
       TERMINATED BY '\n'
       #    If that gets strange results, comment it out and try this:
       #-TERMINATED BY '\r\n'

IGNORE 1 LINES;

```

A moment on file data structure

1. **Loading CSV to SQL forces us to consider how our data is represented in a file**
2. **Excel handles a variety of file storage formats for us - it will guess which is the proper one**
3. **SQL will not**

   It is a power user tool

   Stuff like file format has defaults but if these don't work we must figure things out for ourselves

File data structure: Problem

1. **The file is a string of bytes - in our simple case, ASCII data**

   Unicode? Wrong class

2. **Some of those bytes are data, some are metadata**

   How does MySQL know where one field begins and another ends?

   Where one record ends another begins?

Solved by convention and definition

1. ## Fields "delimited", or "terminated", by commas

   When MySQL sees a comma, it starts a new field

2. ## Fields are also "enclosed" by double quotes

   When MySQL sees a double quote, it considers all that follows a field until it sees another double quote

3. ## Not redundant

   This structure allows us to have commas in our data

   If an enclosure is missing MySQL will just go with the field terminator

   Double quotes in data can be handled by escaping mechanism

   (e.g. \" is not an enclosure) but we haven't time for that

Solved by convention and definition - 2

1. **Lines (e.g. records, rows) terminated by a line termination character(s)**

   In Windows, generally carriage return + line feed, or CR LF

   \r\n

   In Mac and Linux, line feed

   \n

2. **These characters are VERY ANNOYING because we almost never see them**

   Our programs all have facilities for interpreting them for us

3. **We can figure this out or just guess**

NEVER MIND -- JUST DO THIS

1. Create a database
2. Create a table
3. Save as CSV
4. Run this
5. Check results, if bad fiddle with this and try again

```
#   Loads a CSV file, saved from Excel with "Comma Separated Values" option
#   -   not "Windows" csv, not "MS-DOS" csv
#   -   Excel 2008 for Mac, 12.3.5

#   file command output: ASCII text, with CR line terminators

LOAD DATA LOCAL

    #   EDIT THIS PATH:
    INFILE "/Users/gimli/Desktop/from_excel.csv"

    INTO TABLE learning.from_excel

    FIELDS
        TERMINATED BY ','
        ENCLOSED BY '"'

    LINES
       TERMINATED BY '\n'
       #    If that gets strange results, comment it out and try this:
       #-TERMINATED BY '\r\n'

IGNORE 1 LINES;

```

Data Editing

Properly typing data?

```
DROP TABLE IF EXISTS csv_load.temperature;

CREATE TABLE csv_load.temperature (
    date        DATE,
    city        VARCHAR(20),
    temperature INT(2)
);

```

```
+------------+---------------+-------------+
| date       | city          | temperature |
+------------+---------------+-------------+
| 2011-10-01 | San Francisco |          62 |
| 2011-10-01 | New York      |          63 |
| 2011-10-01 | Austin        |          72 |
| 2011-10-02 | San Francisco |          59 |
| 2011-10-02 | New York      |          58 |
+------------+---------------+-------------+

```

- VERY similar to the simple.csv load example, we've just added data typing
- Note that the values for DATE are in YYYY-MM-DD format; if they weren't, we'd have to load as CHAR() and do some data cleaning
- Also, we have to designated a size for our CHAR and INT types
- Any potential problems with our INT type definition?

These operations CHANGE BASE DATA

Prior operations were all outputs

Temporary results, generated to our specification, without affecting the stored data

Changes to stored data are serious

Be very careful with these operations

Be extremely careful in production

INSERT Table Data

Adding President Obama to presidents table

```
INSERT INTO
    sampdb.president

    SET
        last_name    =    'Obama',
        first_name   =    'Barack',
        suffix       =    NULL,
        city         =    'Houston',
        state        =    'TX',
        birth        =    '1961-08-04',
        death        =    NULL

;

```

Alternate INSERT Syntax

1. That entry is wrong, so let's add the right one
2. VALUES provides another syntax for adding data

```
INSERT INTO
    sampdb.president
VALUES  ('Obama', 'Barack', NULL, 'Honolulu', 'HI', '1961-08-04', NULL)
;

```

Editing database data

1. We could also fix the problem with UPDATE

```
UPDATE
    sampdb.president
SET
    state = 'HI',
    city = 'Honolulu'
WHERE
    last_name = 'Obama'
;
```

DELETE -- Removing Records

- Now we have two entries, let's remove one

```
1 DELETE FROM sampdb.president WHERE last_name = 'Obama' LIMIT 1;
2
```

DELETE -- Removing Many Records

1. ## Without the LIMIT, DELETE takes all records

```
1 DELETE FROM sampdb.president WHERE last_name = 'Obama';
2
```

Removing LOTS of Records

Be careful with DELETE!!

```
DELETE FROM sampdb.president;

```

```
+----------+
| COUNT(*) |
+----------+
|        0 |
+----------+

```

Here, not a big deal:

```
INSERT INTO sampdb.president
    VALUES
        ('Washington','George',NULL,'Wakefield','VA','1732-02-22','1799-12-14'),
        ('Adams','John',NULL,'Braintree','MA','1735-10-30','1826-07-04'),
        ('Jefferson','Thomas',NULL,'Albemarle County','VA','1743-04-13','1826-07-04'),
        ('Madison','James',NULL,'Port Conway','VA','1751-03-16','1836-06-28'),
        ('Monroe','James',NULL,'Westmoreland County','VA','1758-04-28','1831-07-04'),
        . . .
;
```

Databases can protect against these mistakes

```
DROP TABLE IF EXISTS sampdb.president;
CREATE TABLE sampdb.president
(
  last_name  VARCHAR(15) NOT NULL,
  first_name VARCHAR(15) NOT NULL,
  suffix     VARCHAR(5) NULL,
  city       VARCHAR(20) NOT NULL,
  state      VARCHAR(2) NOT NULL,
  birth      DATE NOT NULL,
  death      DATE NULL,
  UNIQUE (last_name, first_name)
);

```

UNIQUE provides a data consistency rule

Prevents insertion of a record with a first_name, last_name tuple matching an extant record

Thus prevents addition of data inconsistent with data already entered

"Consistent" != "Right"

If we enter the Hawaii values first, the Illinois ones are prevented

But, we are forced to solve the problem in some way that doesn't risk mistakes like two duplicating records

Summary of Query Components

SELECT

- Specifies the columns output in the query result
- Can be a simple field retrieval from a base table, OR
- An expression of some kind -- function return, mathematical, subquery
- We use SELECT expressions to specify and manipulate output

FROM

- Specifies the source table from which the query draws
- Simplest case is a 'base table' holding data
- BUT because queries themselves always return tables, FROM can refer to another query -- e.g. a subquery
- FROM can also refer to a VIEW -- a shorthand stored on the server referring to a specified query

WHERE

- Filters records in source of query
- Multiple conditions can be specified using AND and OR
- Filters can compare tuples of source record with tuples, and lists of tuples. This allows us to use queries nested in WHERE to specify criteria for inclusion of records from an outer query.
-

GROUP BY

- Specifies subsetting of records for aggregation functions in SELECT
- Records are put into subsets according to their values for the tuple specified in GROUP BY
- The query output will have a row for each unique combination of values for that tuple in the base data

HAVING

- Provides filtration of records output by a query using an aggregrating function
- Output records are filtered by their values -- generally those for some aggregating function

ORDER BY

- Provides control over the order in which query results are displayed
- Can specify order with tuples, each subsquent term ordering records within the order provided by the prior term(s)

JOIN

- Provides means for linking records in one table with those in other tables
- Crucial, as proper data modeling distributes data over a number of tables
- JOIN allows us to recombine those abstracted data into the outputs we want
- Works by creating joined records, among those of specified tables, of those records in those tables whose values in specified fields match
- (Note that JOIN can use comparisons other than equality.)

Primary and Foreign Keys

Primary Keys specify unique records in a table

- Primary key field or fields are specified when table is created
- Database server will not enter data that duplicates primary key values
- Other tables can reference entities in the table by these primary key fields

Foreign keys link a table to a 'parent'

The 'child' table specifies a foreign key at creation

- Foreign keys can be one or many fields
- All entries to the child table must have entries, matching on the foreign key values, in the parent table
- These values need not be unique in the child table
- The foreign key 'links' the child table to the parent
- The presence of parent entries matching child entries is called 'referential integrity
- Database server protects referential integrity by refusing child entries without matching parents
- Server can also be setup to automatically delete all child records of a parent record when that parent record is deleted

Normal Forms

Modeling "norms" protect data clarity

- Poorly organized data models are hard to understand
- Worse, data gets lost, or duplicated
- Creating risk that data become inconsistent with the model
- The problem gets worse as data becomes increasingly involved
- 'Norms' provide rules that prevent many inconsistencies and losses

With all this syntax, we don't care about base table organization

- We are accustomed to Excel-like data organization, where the 'base' tables are some commonly used views
- and we derive other views of the data from those
- But in SQL, we _always_ look at some view of the base data anyway
- We have to split our expectations of how we view the data from the way in which it is organized in base tables
- Base tables should be organized for clarity of the data model, NOT for some particular view of the data

First Normal Form

Data should be atomic, eg values shouldn't be combined

"George Washington" shouldn't be a value, because it combines first and last name

Unless we _never_ will search on or retrieve first and last name

No repeating columns

No columns like author1, author2, author3

Why? We don't dependency on order, nor further processing

Values should be retrievable by name, not by further parsing or processing

We don't want values established by order -- again, we want values retrievable by field name

Second Normal Form

First Normal, plus:

No 'partial dependencies'

Cannot determine a nonkey column value with only a portion of the primary key

Such values should be remodeled to a table where the nonkey values are related to the (new) table's full primary key

Why? We want tables focused on entities

A partial dependency says that there are subentities to the entities in the table

Those should be removed to a table where they are the entities

This avoids duplication of information, confusion over entity definitions, and location of information

Third Normal Form

Second Normal, plus:

No nonkey column depends on the value of another nonkey column

Values are driven by the primary key fields

Again, such values should be removed to a table of their own

Why? Again, we want to avoid subentities

Basic idea: Primary Keys to rule them all

Data should be broken down so that any datum is found by searching on some table on some primary key, and only by that table and primary key

Because primary keys are table-unique, this organizes data so there are no duplications

Note that this can slow performance, because we have to join tables to provide denormalized views

But the model avoids repetition and inconsistency of data

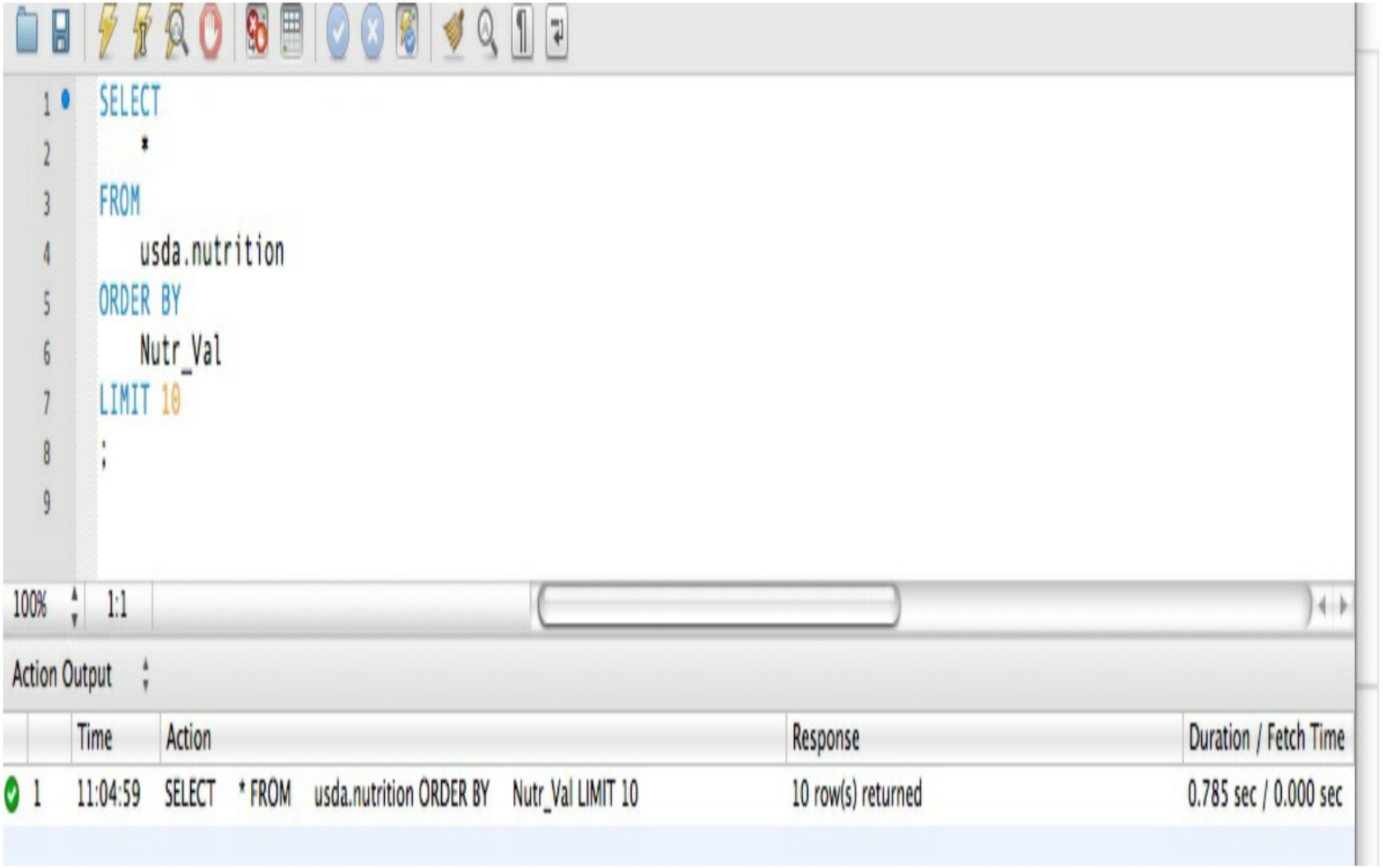
As models become more complex, they outgrow our ability to easily hold them entirely in our heads and notice problems

The normal forms protect us from inconsistent and out of date data

Performance and Indexing

Consider a large table

ORDER BY takes a long time to run on 590,000 records



Indexing makes it faster

An index presorts a specific field on a table

That speeds up search operations enormously

```
ALTER TABLE
    usda.nutrition
ADD INDEX
    nutr_val (Nutr_Val)
;

```

```

```

Faster!

Takes no time at all!

Action Output

	Time	Action	Response	Duration / Fetch Time
1	11:04:59	SELECT * FROM usda.nutrition ORDER BY Nutr_Val LIMIT 10	10 row(s) returned	0.785 sec / 0.000 sec
2	11:23:08	SELECT * FROM usda.nutrition ORDER BY Nutr_Val LIMIT 10	10 row(s) returned	0.000 sec / 0.000 sec

Why indexing works

An index is a pre-ordered sort of a particular field

- Done and stored by the database server
- In an unordered list, the search must check every entry until it hits a match
- In an ordered list, dictionary, the search skips large blocks of entries as before or after the target
- E.g., a dictionary is faster than an unsorted pile of words
- We don't index everything because indexing has storage and performance costs
- For every index on a table, the database must reorder the index for a new entry
- Each index takes up space, in storage and memory

Workflow Integration

How would we chart the Titanic data?

People want graphs, not tables!

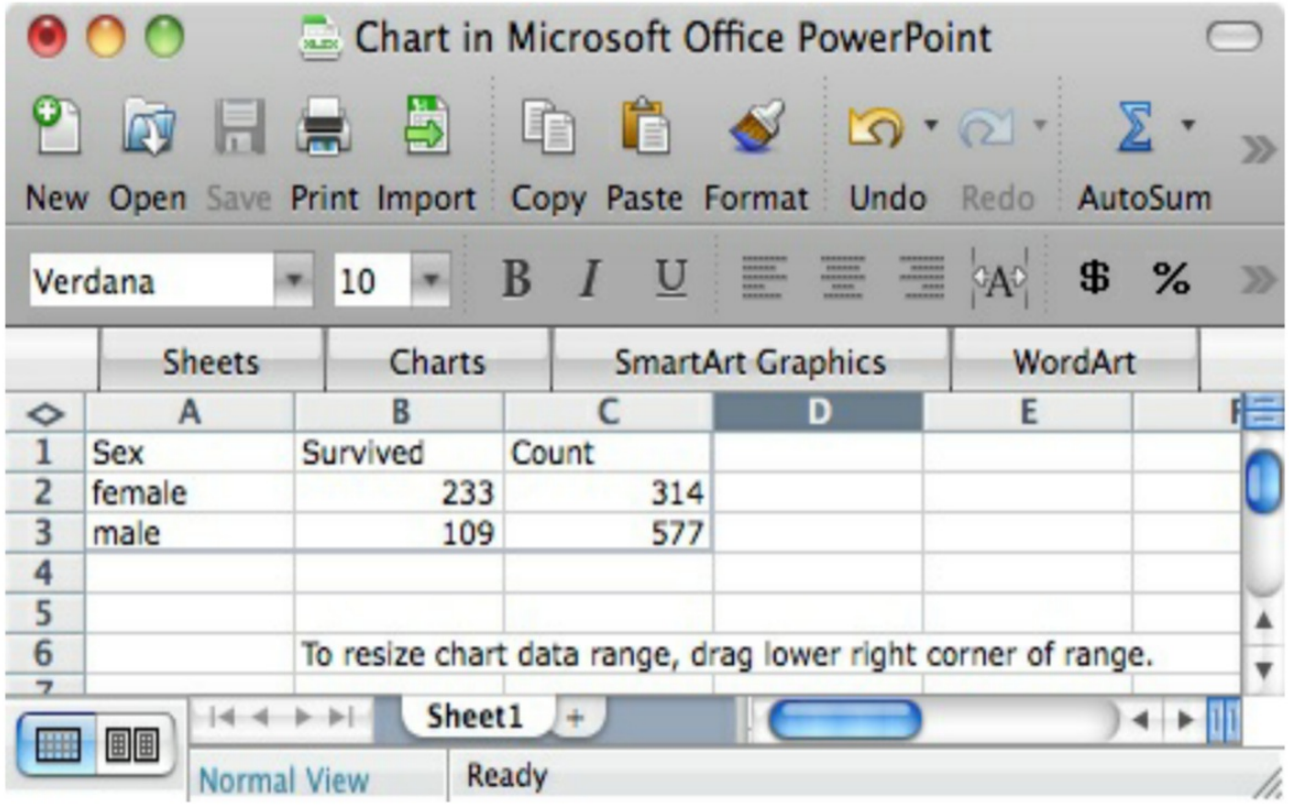
But, SQL doesn't provide vizualization tools

We have to move the data to a program that does

Fortunately, moving data is nothing new

Powerpoint Data Entry

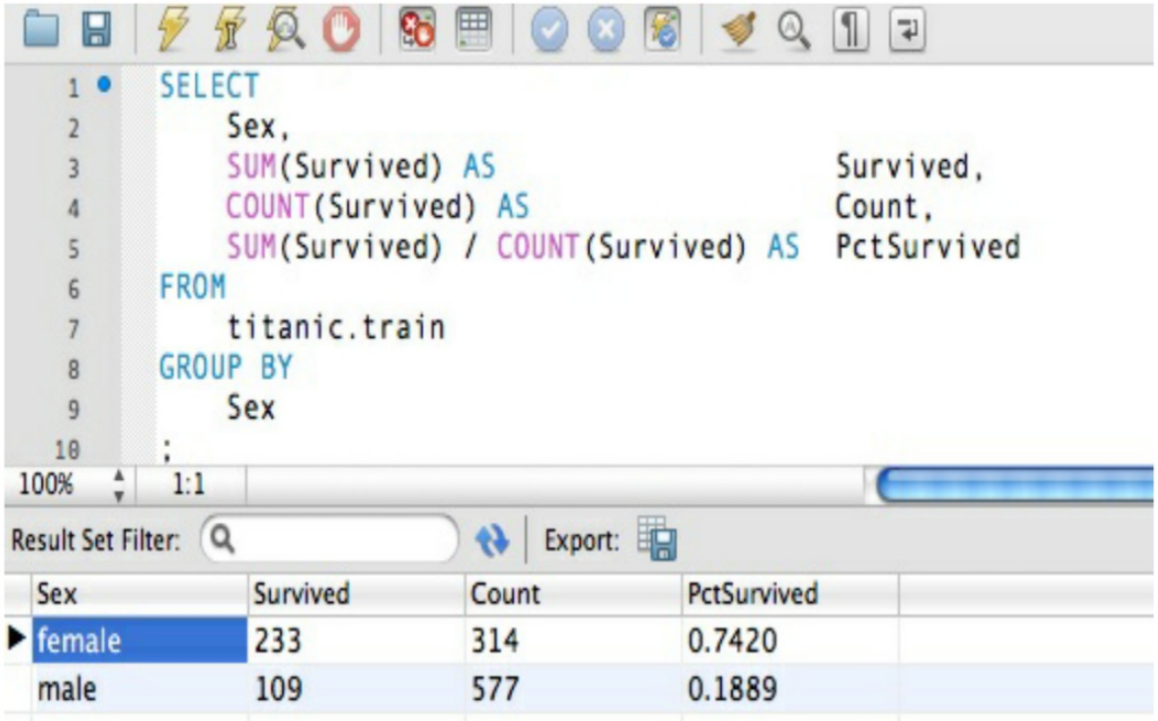
We already paste data from Excel to Powerpoint for graphing



- We can do the same thing from Workbench
- And, many other GUI clients
- Or, from Excel

From Workbench:

Use CSV export



- Note that here the query doesn't specify column headers -- the exported CSV includes a header row
- Clicking 'Export' opens a dialog for saving results to CSV
- These can be opened for copy and paste in Excel -- or directly into the Powerpoint graph

Queries can export direct to CSV

```
-- LACKS COLUMN HEADERS
SELECT
    "Sex",
    "Survived",
    "Count",
    "PctSurvived"

UNION

SELECT
    Sex,
    SUM(Survived) AS                    Survived,
    COUNT(Survived) AS                  Count,
    SUM(Survived) / COUNT(Survived) AS  PctSurvived
FROM
    titanic.train
GROUP BY
    Sex
-- BEWARE permissions issues
INTO OUTFILE
    '/tmp/survival_chart.csv'
FIELDS
    TERMINATED BY ','
    OPTIONALLY ENCLOSED BY '"'
LINES
    TERMINATED BY '\n';
;

```

What is UNION?

SQL statement concatenating two query results into one

- Must have the same number of columns
- Combines the rows of one query with another
- We use this here to put a header row into our exported data

Here, paste serves as a crude 'interface'

An interface is a point where one component provides input to another

- SQL serves as an analytic 'component'
- Powerpoint / D3 / Tableau / et cetera serve as graphing components
- We manually pass the output of our analysis to visualization through copy and paste
- Programming languages (Python, Ruby, PHP) provide direct / automated interface

This manual process is a hassle

- BUT it does mean that we can use ANY visualization tool
- We just have to figure out its interface, and we can write our queries to meet that requirement

Data Cleaning

Google Finance Data

Google NASDAQ:AAPL gordon@practicalhorseshoeing.com + Share

Finance **Apple Inc.** (NASDAQ:AAPL) Add to portfolio More resu

Company
- Summary
- News
- Option chain
- Related companies
- Historical prices
- Financials

Markets

News

Portfolios

Stock screener

Google Domestic Trends

Recent Quotes (Turn on)

You have no recent quotes

Daily prices Jan 31, 2013 - Jan 30, 2014 Update

Date	Open	High	Low	Close	Volume
Jan 29, 2014	503.95	507.37	498.62	500.75	17,991,828
Jan 28, 2014	508.76	515.00	502.07	506.50	38,119,083
Jan 27, 2014	550.07	554.80	545.75	550.50	20,602,736
Jan 24, 2014	554.00	555.62	544.75	546.07	15,483,491
Jan 23, 2014	549.94	556.50	544.81	556.18	14,425,478
Jan 22, 2014	550.91	557.29	547.81	551.51	13,602,762
Jan 21, 2014	540.99	550.07	540.42	549.07	11,750,792
Jan 17, 2014	551.48	552.07	539.90	540.67	15,489,527
Jan 16, 2014	554.90	556.85	551.68	554.25	8,210,190
Jan 15, 2014	553.52	560.20	551.66	557.36	14,067,517
Jan 14, 2014	538.22	546.73	537.66	546.39	11,962,053
Jan 13, 2014	529.91	542.50	529.88	535.73	13,551,549
Jan 10, 2014	539.83	540.80	531.11	532.94	10,902,952
Jan 9, 2014	546.80	546.86	535.35	536.52	9,986,457
Jan 8, 2014	538.81	545.56	538.69	543.46	9,240,955
Jan 7, 2014	544.32	545.96	537.92	540.04	11,347,538

Historical chart

Export

Download to spreadsheet

We can't load this directly

```
DROP TABLE IF EXISTS csv_load.aapl_initial;

CREATE TABLE csv_load.aapl_initial (
    date        VARCHAR(10),
    -- date        DATE,
    open        DECIMAL(5, 2),
    high        DECIMAL(5, 2),
    low         DECIMAL(5, 2),
    close       DECIMAL(5, 2),
    volume      INTEGER(12)

);

```

```
+-----------+--------+--------+--------+--------+----------+
| date      | open   | high   | low    | close  | volume   |
+-----------+--------+--------+--------+--------+----------+
| 29-Jan-14 | 503.95 | 507.37 | 498.62 | 500.75 | 17991828 |
| 28-Jan-14 | 508.76 | 515.00 | 502.07 | 506.50 | 38119083 |
| 27-Jan-14 | 550.07 | 554.80 | 545.75 | 550.50 | 20602736 |
| 24-Jan-14 | 554.00 | 555.62 | 544.75 | 546.07 | 15483491 |
| 23-Jan-14 | 549.94 | 556.50 | 544.81 | 556.18 | 14425478 |
| 22-Jan-14 | 550.91 | 557.29 | 547.81 | 551.51 | 13602762 |
| 21-Jan-14 | 540.99 | 550.07 | 540.42 | 549.07 | 11750792 |
| 17-Jan-14 | 551.48 | 552.07 | 539.90 | 540.67 | 15489527 |
| 16-Jan-14 | 554.90 | 556.85 | 551.68 | 554.25 |  8210190 |
| 15-Jan-14 | 553.52 | 560.20 | 551.66 | 557.36 | 14067517 |
| 14-Jan-14 | 538.22 | 546.73 | 537.66 | 546.39 | 11962053 |
| 13-Jan-14 | 529.91 | 542.50 | 529.88 | 535.73 | 13551549 |
| 10-Jan-14 | 539.83 | 540.80 | 531.11 | 532.94 | 10902952 |
| 9-Jan-14  | 546.80 | 546.86 | 535.35 | 536.52 |  9986457 |
| 8-Jan-14  | 538.81 | 545.56 | 538.69 | 543.46 |  9240955 |
| 7-Jan-14  | 544.32 | 545.96 | 537.92 | 540.04 | 11347538 |
| 6-Jan-14  | 537.45 | 546.80 | 533.60 | 543.93 | 14765593 |
| 3-Jan-14  | 552.86 | 553.70 | 540.43 | 540.98 | 14043410 |
| 2-Jan-14  | 555.68 | 557.03 | 552.02 | 553.13 |  8398851 |
| 31-Dec-13 | 554.17 | 561.28 | 554.00 | 561.02 |  7974196 |
+-----------+--------+--------+--------+--------+----------+

```

- We can't load date data in this format

We can parse the data

```
DROP VIEW IF EXISTS csv_load.date_parsed;

CREATE VIEW csv_load.date_parsed

AS

SELECT
    date,
    LEFT(date, LOCATE('-', date) - 1 ) as day,
    RIGHT(date, LENGTH(date) - LOCATE('-', date, 4) ) as yr,
    MID(
        date,
        LOCATE('-', date) + 1,
        ( LOCATE('-', date, 4) - LOCATE('-', date) - 1 )
    ) as mnt,
    open,
    high,
    low,
    close,
    volume

FROM
    csv_load.aapl_initial
;
```

date	day	yr	mnt	open	high	low	close	volume
29-Jan-14	29	14	Jan	503.95	507.37	498.62	500.75	17991828
28-Jan-14	28	14	Jan	508.76	515.00	502.07	506.50	38119083
27-Jan-14	27	14	Jan	550.07	554.80	545.75	550.50	20602736
24-Jan-14	24	14	Jan	554.00	555.62	544.75	546.07	15483491
23-Jan-14	23	14	Jan	549.94	556.50	544.81	556.18	14425478
22-Jan-14	22	14	Jan	550.91	557.29	547.81	551.51	13602762
21-Jan-14	21	14	Jan	540.99	550.07	540.42	549.07	11750792
17-Jan-14	17	14	Jan	551.48	552.07	539.90	540.67	15489527
16-Jan-14	16	14	Jan	554.90	556.85	551.68	554.25	8210190
15-Jan-14	15	14	Jan	553.52	560.20	551.66	557.36	14067517
14-Jan-14	14	14	Jan	538.22	546.73	537.66	546.39	11962053
13-Jan-14	13	14	Jan	529.91	542.50	529.88	535.73	13551549
10-Jan-14	10	14	Jan	539.83	540.80	531.11	532.94	10902952
9-Jan-14	9	14	Jan	546.80	546.86	535.35	536.52	9986457
8-Jan-14	8	14	Jan	538.81	545.56	538.69	543.46	9240955
7-Jan-14	7	14	Jan	544.32	545.96	537.92	540.04	11347538
6-Jan-14	6	14	Jan	537.45	546.80	533.60	543.93	14765593
3-Jan-14	3	14	Jan	552.86	553.70	540.43	540.98	14043410
2-Jan-14	2	14	Jan	555.68	557.03	552.02	553.13	8398851
31-Dec-13	31	13	Dec	554.17	561.28	554.00	561.02	7974196

- We use string functions to break up -- parse -- the date string

We can reformat the data

```
DROP VIEW IF EXISTS csv_load.date_reformat;

CREATE VIEW
    csv_load.date_reformat
AS
SELECT
    CONCAT(
        -- year term
        CONCAT( '20', yr),
        '-',
        -- month term
        CASE
            WHEN mnt = 'Jan' THEN '01'
            WHEN mnt = 'Feb' THEN '02'
            WHEN mnt = 'Mar' THEN '03'
            WHEN mnt = 'Apr' THEN '04'
            WHEN mnt = 'May' THEN '05'
            WHEN mnt = 'Jun' THEN '06'
            WHEN mnt = 'Jul' THEN '07'
            WHEN mnt = 'Aug' THEN '08'
            WHEN mnt = 'Sep' THEN '09'
            WHEN mnt = 'Oct' THEN '10'
            WHEN mnt = 'Nov' THEN '11'
            WHEN mnt = 'Dec' THEN '12'
        END,
        '-',
        -- day term
        CONCAT( IF(LENGTH(day) = 1, '0', ''), day
        )
    ) AS date,
    open,
    high,
    low,
    close,
    volume

FROM
    csv_load.date_parsed
;
```

```
+------------+--------+--------+--------+--------+----------+
| date       | open   | high   | low    | close  | volume   |
+------------+--------+--------+--------+--------+----------+
| 2014-01-29 | 503.95 | 507.37 | 498.62 | 500.75 | 17991828 |
| 2014-01-28 | 508.76 | 515.00 | 502.07 | 506.50 | 38119083 |
| 2014-01-27 | 550.07 | 554.80 | 545.75 | 550.50 | 20602736 |
| 2014-01-24 | 554.00 | 555.62 | 544.75 | 546.07 | 15483491 |
| 2014-01-23 | 549.94 | 556.50 | 544.81 | 556.18 | 14425478 |
| 2014-01-22 | 550.91 | 557.29 | 547.81 | 551.51 | 13602762 |
| 2014-01-21 | 540.99 | 550.07 | 540.42 | 549.07 | 11750792 |
| 2014-01-17 | 551.48 | 552.07 | 539.90 | 540.67 | 15489527 |
| 2014-01-16 | 554.90 | 556.85 | 551.68 | 554.25 |  8210190 |
| 2014-01-15 | 553.52 | 560.20 | 551.66 | 557.36 | 14067517 |
| 2014-01-14 | 538.22 | 546.73 | 537.66 | 546.39 | 11962053 |
| 2014-01-13 | 529.91 | 542.50 | 529.88 | 535.73 | 13551549 |
| 2014-01-10 | 539.83 | 540.80 | 531.11 | 532.94 | 10902952 |
| 2014-01-09 | 546.80 | 546.86 | 535.35 | 536.52 |  9986457 |
| 2014-01-08 | 538.81 | 545.56 | 538.69 | 543.46 |  9240955 |
| 2014-01-07 | 544.32 | 545.96 | 537.92 | 540.04 | 11347538 |
| 2014-01-06 | 537.45 | 546.80 | 533.60 | 543.93 | 14765593 |
| 2014-01-03 | 552.86 | 553.70 | 540.43 | 540.98 | 14043410 |
| 2014-01-02 | 555.68 | 557.03 | 552.02 | 553.13 |  8398851 |
| 2013-12-31 | 554.17 | 561.28 | 554.00 | 561.02 |  7974196 |
+------------+--------+--------+--------+--------+----------+
```

- We use CASE to transpose string months to digits
- and CONCAT to form a proper date string for load as DATE

This, we can load

```
INSERT INTO
    csv_load.aapl
SELECT
    date,
    open,
    high,
    low,
    close,
    volume
FROM
    csv_load.date_reformat
;

```

```
+------------+--------+--------+--------+--------+----------+
| date       | open   | high   | low    | close  | volume   |
+------------+--------+--------+--------+--------+----------+
| 2014-01-29 | 503.95 | 507.37 | 498.62 | 500.75 | 17991828 |
| 2014-01-28 | 508.76 | 515.00 | 502.07 | 506.50 | 38119083 |
| 2014-01-27 | 550.07 | 554.80 | 545.75 | 550.50 | 20602736 |
| 2014-01-24 | 554.00 | 555.62 | 544.75 | 546.07 | 15483491 |
| 2014-01-23 | 549.94 | 556.50 | 544.81 | 556.18 | 14425478 |
| 2014-01-22 | 550.91 | 557.29 | 547.81 | 551.51 | 13602762 |
| 2014-01-21 | 540.99 | 550.07 | 540.42 | 549.07 | 11750792 |
| 2014-01-17 | 551.48 | 552.07 | 539.90 | 540.67 | 15489527 |
| 2014-01-16 | 554.90 | 556.85 | 551.68 | 554.25 |  8210190 |
| 2014-01-15 | 553.52 | 560.20 | 551.66 | 557.36 | 14067517 |
| 2014-01-14 | 538.22 | 546.73 | 537.66 | 546.39 | 11962053 |
| 2014-01-13 | 529.91 | 542.50 | 529.88 | 535.73 | 13551549 |
| 2014-01-10 | 539.83 | 540.80 | 531.11 | 532.94 | 10902952 |
| 2014-01-09 | 546.80 | 546.86 | 535.35 | 536.52 |  9986457 |
| 2014-01-08 | 538.81 | 545.56 | 538.69 | 543.46 |  9240955 |
| 2014-01-07 | 544.32 | 545.96 | 537.92 | 540.04 | 11347538 |
| 2014-01-06 | 537.45 | 546.80 | 533.60 | 543.93 | 14765593 |
| 2014-01-03 | 552.86 | 553.70 | 540.43 | 540.98 | 14043410 |
| 2014-01-02 | 555.68 | 557.03 | 552.02 | 553.13 |  8398851 |
| 2013-12-31 | 554.17 | 561.28 | 554.00 | 561.02 |  7974196 |
+------------+--------+--------+--------+--------+----------+
```

- This string loads properly as DATE

Now we can do date based analysis

```
SELECT
    MIN(date) as week_start,
    MAX(high) AS weekly_high,
    MIN(low) AS weekly_low
FROM
    csv_load.aapl
GROUP BY
    week(date)
ORDER BY
    week_start
LIMIT
    20
;

```

```
+------------+-------------+------------+
| week_start | weekly_high | weekly_low |
+------------+-------------+------------+
| 2013-02-01 |      554.80 |     448.35 |
| 2013-02-04 |      478.81 |     442.00 |
| 2013-02-11 |      484.94 |     459.92 |
| 2013-02-19 |      462.73 |     442.82 |
| 2013-02-25 |      455.12 |     429.98 |
| 2013-03-04 |      435.43 |     419.00 |
| 2013-03-11 |      444.23 |     425.14 |
| 2013-03-18 |      462.10 |     441.20 |
| 2013-03-25 |      469.95 |     441.62 |
| 2013-04-01 |      443.70 |     419.68 |
| 2013-04-08 |      437.99 |     422.49 |
| 2013-04-15 |      427.89 |     385.10 |
| 2013-04-22 |      418.77 |     391.28 |
| 2013-04-29 |      453.23 |     420.00 |
| 2013-05-06 |      465.75 |     450.48 |
| 2013-05-13 |      457.90 |     418.90 |
| 2013-05-20 |      448.35 |     430.10 |
| 2013-05-28 |      457.10 |     439.40 |
| 2013-06-03 |      454.43 |     432.77 |
| 2013-06-10 |      449.08 |     428.50 |
+------------+-------------+------------+
```

- WEEK() returns which week of the year a date falls within
- We can use this value to group our data, and extract weekly highs and lows

Data Analysis -- More

Let's reprise data loading

We have this CSV data on Titanic survivors

```
PassengerId,Survived,Pclass,Name,Sex,Age,SibSp,Parch,Ticket,Fare,Cabin,Embarked
1,0,3,"Braund, Mr. Owen Harris",male,22,1,0,A/5 21171,7.25,,S
2,1,1,"Cumings, Mrs. John Bradley (Florence Briggs Thayer)",female,38,1,0,PC 17599,71.2833,C85,C
3,1,3,"Heikkinen, Miss. Laina",female,26,0,0,STON/O2. 3101282,7.925,,S
4,1,1,"Futrelle, Mrs. Jacques Heath (Lily May Peel)",female,35,1,0,113803,53.1,C123,S
5,0,3,"Allen, Mr. William Henry",male,35,0,0,373450,8.05,,S
6,0,3,"Moran, Mr. James",male,,0,0,330877,8.4583,,Q
7,0,1,"McCarthy, Mr. Timothy J",male,54,0,0,17463,51.8625,E46,S
8,0,3,"Palsson, Master. Gosta Leonard",male,2,3,1,349909,21.075,,S
9,1,3,"Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)",female,27,0,2,347742,11.1333,,S
11
```

Create database, table

Familiar?

- These tell the database server to create containers for the data
- We name the table 'train' as part of an analytic technique -- 'training' a model on a subset of the available data

```
DROP DATABASE IF EXISTS titanic;
CREATE DATABASE titanic;

```

```
DROP TABLE IF EXISTS titanic.train;
CREATE TABLE titanic.train (
    PassengerId INT(4),
    Survived    INT(1),
    Pclass      INT(3),
    Name        VARCHAR(50),
    Sex         CHAR(6),
    Age         INT(3),
    SibSp       INT(2),
    Parch       VARCHAR(50),
    Ticket      INT(5),
    Fare        DECIMAL(10,2),
    Cabin       VARCHAR(10),
    Embarked    CHAR(2)
);

```

Load the data

Familiar?

```
LOAD DATA LOCAL

    #   EDIT THIS PATH:
    INFILE "/Users/gimli/Work/teaching/practical_sql/class_materials/deck_templates/december2013/class_04/titanic_setup/data/train.csv"

    INTO TABLE titanic.train

    FIELDS
        TERMINATED BY ','
        ENCLOSED BY '"'

    LINES
       #-TERMINATED BY '\n'
       #    If that gets strange results, comment it out and try this:
       TERMINATED BY '\r\n'

IGNORE 1 LINES;

```

This tells the server where to find the file, where to put the data, and how to identify particular fields and records

Now we have a table

```
DESCRIBE titanic.train;
```

```
+-------------+---------------+------+-----+---------+-------+
| Field       | Type          | Null | Key | Default | Extra |
+-------------+---------------+------+-----+---------+-------+
| PassengerId | int(4)        | YES  |     | NULL    |       |
| Survived    | int(1)        | YES  |     | NULL    |       |
| Pclass      | int(3)        | YES  |     | NULL    |       |
| Name        | varchar(50)   | YES  |     | NULL    |       |
| Sex         | char(6)       | YES  |     | NULL    |       |
| Age         | int(3)        | YES  |     | NULL    |       |
| SibSp       | int(2)        | YES  |     | NULL    |       |
| Parch       | varchar(50)   | YES  |     | NULL    |       |
| Ticket      | int(5)        | YES  |     | NULL    |       |
| Fare        | decimal(10,2) | YES  |     | NULL    |       |
| Cabin       | varchar(10)   | YES  |     | NULL    |       |
| Embarked    | char(2)       | YES  |     | NULL    |       |
+-------------+---------------+------+-----+---------+-------+
```

And the table holds data

```
SELECT * FROM titanic.train LIMIT 10;
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Far
1	0	3	Braund, Mr. Owen Harris	male	22	1	0	0	7.
2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Thayer	female	38	1	0	0	71.
3	1	3	Heikkinen, Miss. Laina	female	26	0	0	0	7.
4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35	1	0	113803	53.
5	0	3	Allen, Mr. William Henry	male	35	0	0	373450	8.
6	0	3	Moran, Mr. James	male	0	0	0	330877	8.
7	0	1	McCarthy, Mr. Timothy J	male	54	0	0	17463	51.
8	0	3	Palsson, Master. Gosta Leonard	male	2	3	1	349909	21.
9	1	3	Johnson, Mrs. Oscar W (Elisabeth Vilhelmina Berg)	female	27	0	2	347742	11.
10	1	2	Nasser, Mrs. Nicholas (Adele Achem)	female	14	1	0	237736	30.

Analysing Data

Let's explore that

Use DISTINCT and GROUP BY to see what's there

```
SELECT DISTINCT
    Survived,
    Pclass,
    Sex,
    Parch,
    Embarked
FROM
    titanic.train
LIMIT 30
;
```

Survived	Pclass	Sex	Parch	Embarked
0	3	male	0	S
1	1	female	0	C
1	3	female	0	S
1	1	female	0	S
0	3	male	0	Q
0	1	male	0	S
0	3	male	1	S
1	3	female	2	S
1	2	female	0	C
1	3	female	1	S
0	3	male	5	S
0	3	female	0	S
1	2	female	0	S
0	3	male	1	Q
1	2	male	0	S
1	3	female	0	C
0	2	male	0	S
1	3	female	0	Q
1	1	male	0	S
0	3	female	1	S
1	3	female	5	S
0	3	male	0	C
0	1	male	2	S
0	1	male	0	C
1	3	male	0	C
0	2	female	0	S
1	2	female	2	C
0	1	male	1	C
1	2	female	2	S
0	3	male	2	S

Who survived?

We use GROUP BY and output transformation

```
SELECT
    Sex,
    SUM(Survived) AS                  Survived,
    COUNT(Survived) AS                Count,
    SUM(Survived) / COUNT(Survived) AS  PctSurvived
FROM
    titanic.train
GROUP BY
    Sex
;

```

```
+--------+----------+-------+-------------+
| Sex    | Survived | Count | PctSurvived |
+--------+----------+-------+-------------+
| female |      233 |   314 |      0.7420 |
| male   |      109 |   577 |      0.1889 |
+--------+----------+-------+-------------+

```

We can sort the fares into bins

Use CASE to assign fare values into "buckets"

```
DROP VIEW IF EXISTS titanic.binned;
CREATE VIEW titanic.binned
AS
    SELECT
        *,
        CASE
            WHEN Fare <= 50
                THEN 1
            WHEN Fare > 50 AND Fare <= 100
                THEN 2
            WHEN Fare > 100 AND Fare <= 150
                THEN 3
            WHEN Fare > 150 AND Fare <= 200
                THEN 4
            ELSE
                5
            END AS FareBin
    FROM
        titanic.train
;

```

- CASE allows us to assign values to tiers

Now we've a VIEW with bins

```
SELECT
    PassengerId,
    Fare,
    FareBin
FROM
    titanic.binned
LIMIT 10
;

```

```
+-------------+-------+---------+
| PassengerId | Fare  | FareBin |
+-------------+-------+---------+
|           1 |  7.25 |       1 |
|           2 | 71.28 |       2 |
|           3 |  7.93 |       1 |
|           4 | 53.10 |       2 |
|           5 |  8.05 |       1 |
|           6 |  8.46 |       1 |
|           7 | 51.86 |       2 |
|           8 | 21.08 |       1 |
|           9 | 11.13 |       1 |
|          10 | 30.07 |       1 |
+-------------+-------+---------+

```

Now we can add bins to our survival analysis

```
SELECT
                                    Sex,
                                    FareBin,
    SUM(Survived) AS                SumSurvived,
    COUNT(Survived) AS              CountSurvived,
    SUM(Survived) / COUNT(Survived) AS  PctSurvived
FROM
    titanic.binned
GROUP BY
    Sex,
    FareBin
ORDER BY
    Sex,
    FareBin
;


```

```
+--------+---------+-------------+---------------+-------------+
| Sex    | FareBin | SumSurvived | CountSurvived | PctSurvived |
+--------+---------+-------------+---------------+-------------+
| female |       1 |         151 |           227 |      0.6652 |
| female |       2 |          50 |            53 |      0.9434 |
| female |       3 |          15 |            15 |      1.0000 |
| female |       4 |           5 |             7 |      0.7143 |
| female |       5 |          12 |            12 |      1.0000 |
| male   |       1 |          82 |           504 |      0.1627 |
| male   |       2 |          20 |            54 |      0.3704 |
| male   |       3 |           4 |             9 |      0.4444 |
| male   |       4 |           1 |             2 |      0.5000 |
| male   |       5 |           2 |             8 |      0.2500 |
+--------+---------+-------------+---------------+-------------+

```

We can add further "dimensions"

```
SELECT
                                    Sex,
                                    FareBin,
                                    Embarked,
    SUM(Survived) AS                SumSurvived,
    COUNT(Survived) AS              CountSurvived,
    SUM(Survived) / COUNT(Survived) AS  PctSurvived
FROM
    titanic.binned
GROUP BY
    Sex,
    FareBin,
    Embarked
ORDER BY
    Sex,
    FareBin,
    Embarked
;


```

Digging deeper into the data

Sex	FareBin	Embarked	SumSurvived	CountSurvived	PctSurvived
female	1	C	27	36	0.7500
female	1	Q	26	35	0.7429
female	1	S	98	156	0.6282
female	2		2	2	1.0000
female	2	C	19	19	1.0000
female	2	Q	1	1	1.0000
female	2	S	28	31	0.9032
female	3	C	11	11	1.0000
female	3	S	4	4	1.0000
female	4	S	5	7	0.7143
female	5	C	7	7	1.0000
female	5	S	5	5	1.0000
male	1	C	18	71	0.2535
male	1	Q	3	40	0.0750
male	1	S	61	393	0.1552
male	2	C	8	13	0.6154
male	2	Q	0	1	0.0000
male	2	S	12	40	0.3000
male	3	C	1	6	0.1667
male	3	S	3	3	1.0000
male	4	S	1	2	0.5000
male	5	C	2	5	0.4000
male	5	S	0	3	0.0000

SQL allows more "dimensions" than Excel

- We can imagine a 2 x 2 pivot table, and even a third dimension
- But SQL allows us to subdivide the data on as many fields as we can imagine
- We can use this to simply filter out records (e.g., let's look only at females in fare bin 3
- Or, to look outcomes based on subcategories
- We could also write queries calculating simple linear regression values

Summing Up

Takeaways

- Model is "verbal", not visual -- which requires adjustment from Excel based habits
- Syntax is 'Germanic', expressions later in queries can change the effect of preceding statements
- Practice builds intuition and conceptual orientation

Takeaways

- Base data is immutable and not changed -- we use queries to describe desired outputs
- Notice that this frees us to arrange the base data in ways most amenable to query, rather than according to some particular view
- What matters about base table arrangement is our ability to describe outputs from it, rather than its immediate / initial appearance
- We conduct more complex analysis by performing queries on output of queries, eg subqueries and joins
- Mastering the basic syntax allows us to devise more complicated expressions, passing one analysis output on to another

Get the Deck

gordon@practicalhorseshoeing.com