Lecture #12: SQL
Btw. CNN criticizing my work...

Question

Next lecture…

A) Q&A Session
B) Fun Lecture
C) Review Lecture
D) All of the above
Database Management Systems

Database Management System

Applications
Other DBMS
Users

DBMS
(Define, record, query, update, manage data)

Storage Area

Relational Database
Hierarchical Database
Flat Files Database
Objects Database
The SQL language is represented in query strings delivered to a DB backend.

Use the techniques learned here to build clean abstractions.

You have already learned the relational operators!
Data 8 Tables

- A single, simple, powerful data structure for all
- Inspired by Excel, SQL, R, Pandas, Numpy, …

- An ordered collection of labeled columns of anything
- Inspired by Excel, SQL, R, Pandas, Numpy, …
Database Management Systems

• DBMS are persistent tables with powerful relational operators
  – Important, heavily used, interesting!

• A table is a collection of records, which are rows that have a value for each column

- Structure Query Language (SQL) is a declarative programming language describing operations on tables

<table>
<thead>
<tr>
<th>Name</th>
<th>Latitude</th>
<th>Longitude</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berkeley</td>
<td>38</td>
<td>122</td>
</tr>
<tr>
<td>Cambridge</td>
<td>42</td>
<td>71</td>
</tr>
<tr>
<td>Minneapolis</td>
<td>45</td>
<td>93</td>
</tr>
</tbody>
</table>

- row has a value for each column
- column has a name and a type
- table has columns and rows
SQL

• A declarative language
  – Described what to compute
  – Imperative languages, like python, describe how to compute it
  – Query processor (interpreter) chooses which of many equivalent query plans to execute to perform the SQL statements

• ANSI and ISO standard, but many variants

• select statement creates a new table, either from scratch or by projecting a table

• create table statement gives a global name to a table

• Lots of other statements
  – analyze, delete, explain, insert, replace, update, …

• The action is in select
SQL example

- SQL statements create tables
  - Give it a try with sqlite3 or http://kripken.github.io/sql.js/GUI/
  - Each statement ends with ‘;’

```bash
$ sqlite3
SQLite version 3.9.2 2015-11-02 18:31:45
Enter "help" for usage hints.
Connected to a transient in-memory database.
Use "open FILENAME" to reopen on a persistent database.
sqlite> select 38 as latitude, 122 as longitude, "Berkeley" as name;
38|122|Berkeley
sqlite>
```
# An example of creating a Table from a list of rows.
Table(["Flavor", "Color", "Price"]).with_rows([
('strawberry', 'pink', 3.55),
('chocolate', 'light brown', 4.75),
('chocolate', 'dark brown', 5.25),
('strawberry', 'pink', 5.25),
('bubblegum', 'pink', 4.75)])

<table>
<thead>
<tr>
<th>Flavor</th>
<th>Color</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>strawberry</td>
<td>pink</td>
<td>3.55</td>
</tr>
<tr>
<td>chocolate</td>
<td>light brown</td>
<td>4.75</td>
</tr>
<tr>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
<tr>
<td>strawberry</td>
<td>pink</td>
<td>5.25</td>
</tr>
<tr>
<td>bubblegum</td>
<td>pink</td>
<td>4.75</td>
</tr>
</tbody>
</table>
select

• Comma-separated list of *column descriptions*
• Column description is an expression, optionally followed by *as* and a *column name*

```
select [expression] as [name], [expression] as [name]; ...
```

• Selecting literals creates a one-row table

```
select "strawberry" as Flavor, "pink" as Color, 3.55 as Price;
```

• *union* of select statements is a table containing the union of the rows

```
select "strawberry" as Flavor, "pink" as Color, 3.55 as Price union
select "chocolate","light brown", 4.75 union
select "chocolate","dark brown", 5.25 union
select "strawberry","pink",5.25 union
select "bubblegum","pink",4.75;
```
create table

• SQL often used interactively
  – Result of select displayed to the user, but not stored
• Create table statement gives the result a name
  – Like a variable, but for a permanent object

create table [name] as [select statement];
create table cones as
    select 1 as ID, "strawberry" as Flavor, "pink" as Color, 3.55 as Price union
    select 2, "chocolate","light brown", 4.75 union
    select 3, "chocolate","dark brown", 5.25 union
    select 4, "strawberry","pink",5.25 union
    select 5, "bubblegum","pink",4.75 union
    select 6, "chocolate", "dark brown", 5.25;

Notice how column names are introduced and implicit later on.
Select ...

```sql
[culler@CullerMac ~/Classes/CS88-Fa18/ideas/sql> sqlite3 icecream.db
SQLite version 3.13.0 2016-05-18 10:57:30
Enter ".help" for usage hints.
sqlite> create table cones as
    ...>   select 1 as ID, "strawberry" as Flavor, "pink" as Color, 3.55 as Price
    ...>   select 2, "chocolate","light brown", 4.75 union
    ...>   select 3, "chocolate","dark brown", 5.25 union
    ...>   select 4, "strawberry","pink",5.25 union
    ...>   select 5, "bubblegum","pink",4.75 union
    [ ...>   select 6, "chocolate", "dark brown", 5.25;

sqlite> select * from cones;
1|strawberry|pink|3.55
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
5|bubblegum|pink|4.75
6|chocolate|dark brown|5.25
sqlite>
```

```
cones = Table(["ID", "Flavor","Color","Price"]).with_rows(  
    (1, 'strawberry', 'pink', 3.55),
    (2, 'chocolate', 'light brown', 4.75),
    (3, 'chocolate', 'dark brown', 5.25),
    (4, 'strawberry', 'pink', 5.25),
    (5, 'bubblegum', 'pink', 4.75),
    (6, 'chocolate', 'dark brown', 5.25)
)

<table>
<thead>
<tr>
<th></th>
<th>Flavour</th>
<th>Color</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>strawberry</td>
<td>pink</td>
<td>3.55</td>
</tr>
<tr>
<td>2</td>
<td>chocolate</td>
<td>light brown</td>
<td>4.75</td>
</tr>
<tr>
<td>3</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
<tr>
<td>4</td>
<td>strawberry</td>
<td>pink</td>
<td>5.25</td>
</tr>
<tr>
<td>5</td>
<td>bubblegum</td>
<td>pink</td>
<td>4.75</td>
</tr>
<tr>
<td>6</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
</tbody>
</table>
```
Projecting existing tables

- Input table specified by \texttt{from} clause
- Subset of rows selected using a \texttt{where} clause
- Ordering of the selected rows declared using an \texttt{order by} clause

```
select [columns] from [table] where [condition] order by [order];
```

```
select * from cones order by Price;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Flavor</th>
<th>Color</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>strawberry</td>
<td>pink</td>
<td>3.55</td>
</tr>
<tr>
<td>2</td>
<td>chocolate</td>
<td>light brown</td>
<td>4.75</td>
</tr>
<tr>
<td>5</td>
<td>bubblegum</td>
<td>pink</td>
<td>4.75</td>
</tr>
<tr>
<td>3</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
<tr>
<td>4</td>
<td>strawberry</td>
<td>pink</td>
<td>5.25</td>
</tr>
<tr>
<td>6</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
</tbody>
</table>
Projection

In [5]: cones.select(['Flavor', 'Price'])

Out[5]:
<table>
<thead>
<tr>
<th>Flavor</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>strawberry</td>
<td>3.55</td>
</tr>
<tr>
<td>chocolate</td>
<td>4.75</td>
</tr>
<tr>
<td>chocolate</td>
<td>5.25</td>
</tr>
<tr>
<td>strawberry</td>
<td>5.25</td>
</tr>
<tr>
<td>bubblegum</td>
<td>4.75</td>
</tr>
<tr>
<td>chocolate</td>
<td>5.25</td>
</tr>
</tbody>
</table>

sqlite> select Flavor, Price from cones;
Flavor|Price
strawberry|3.55
chocolate|4.75
chocolate|5.25
strawberry|5.25
bubblegum|4.75
chocolate|5.25

- Select versus indexing a column?
Permanent Data Storage

<table>
<thead>
<tr>
<th>ID</th>
<th>Flavor</th>
<th>Color</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>strawberry</td>
<td>pink</td>
<td>3.55</td>
</tr>
<tr>
<td>2</td>
<td>chocolate</td>
<td>light brown</td>
<td>4.75</td>
</tr>
<tr>
<td>5</td>
<td>bubblegum</td>
<td>pink</td>
<td>4.75</td>
</tr>
<tr>
<td>3</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
<tr>
<td>4</td>
<td>strawberry</td>
<td>pink</td>
<td>5.25</td>
</tr>
<tr>
<td>6</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
</tbody>
</table>
Filtering rows - where

- Set of Table records (rows) that satisfy a condition

```sql
select [columns] from [table] where [condition] order by [order];
```
SQL Operators for predicate

- use the **WHERE** clause in the SQL statements such as **SELECT**, **UPDATE** and **DELETE** to filter rows that do not meet a specified condition.

SQLite understands the following binary operators, in order from highest to lowest precedence:

```plaintext
| | * / % + - << >> & |
< <= > >= == != <> IS IS NOT IN LIKE GLOB MATCH REGEXP
AND
OR

Supported unary prefix operators are these:

- + - NOT
Approximate Matching …

Regular expression matches are so common that they are built in in SQL.

```
sqlite> select * from cones where Flavor like "%berry%"
Flavor|Color|Price
strawberry|pink|3.55
strawberry|pink|5.25
sqlite>
```

On the other hand, you have the full power of Python to express what you mean.

```python
cones.where(cones.apply(lambda x: 'berry' in x, 'Flavor'))
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Flavor</th>
<th>Color</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>strawberry</td>
<td>pink</td>
<td>3.55</td>
</tr>
<tr>
<td>4</td>
<td>strawberry</td>
<td>pink</td>
<td>5.25</td>
</tr>
</tbody>
</table>
Group and Aggregate

- The **GROUP BY** clause is used to group rows returned by **SELECT statement** into a set of summary rows or groups based on values of columns or expressions.
- Apply an **aggregate function**, such as **SUM**, **AVG**, **MIN**, **MAX** or **COUNT**, to each group to output the summary information.

<table>
<thead>
<tr>
<th>Flavor</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubblegum</td>
<td>1</td>
</tr>
<tr>
<td>chocolate</td>
<td>3</td>
</tr>
<tr>
<td>strawberry</td>
<td>2</td>
</tr>
</tbody>
</table>

```sql
sqlite> select count(Price), Flavor from cones group by Flavor;
count(Price)|Flavor
1|bubblegum
2|chocolate
2|strawberry
```

```sql
sqlite> select avg(Price), Flavor from cones group by Flavor;
avg(Price)|Flavor
4.75|bubblegum
5.08333|chocolate
4.4|strawberry
```
Unique / Distinct values

```
select DISTINCT [columns] from [table] where [condition] order by [order];
```

```
[sqlite]> select distinct Flavor, Color from cones;
strawberry|pink
chocolate|light brown
chocolate|dark brown
bubblegum|pink
sqlite>
```

```
In [8]: cones.groups(['Flavor', 'Color']).drop('count')
Out[8]:
<table>
<thead>
<tr>
<th>Flavor</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubblegum</td>
<td>pink</td>
</tr>
<tr>
<td>chocolate</td>
<td>dark brown</td>
</tr>
<tr>
<td>chocolate</td>
<td>light brown</td>
</tr>
<tr>
<td>strawberry</td>
<td>pink</td>
</tr>
</tbody>
</table>
```

```
In [7]: np.unique(cones['Flavor'])
Out[7]: array(['bubblegum', 'chocolate', 'strawberry'], dtype='<U10')
```

- Built in to the language or a composable tool?
Joining tables

- Two tables are joined by a comma to yield all combinations of a row from each
  
  `-- select * from sales, cones;`

```sql
sqlite> select * from sales, cones;
Baskin|1|strawberry|pink|3.55
Baskin|2|chocolate|light brown|4.75
Baskin|3|chocolate|dark brown|5.25
Baskin|4|strawberry|pink|5.25
Baskin|5|bubblegum|pink|4.75
Baskin|6|chocolate|dark brown|5.25
Baskin|7|strawberry|pink|3.55
Baskin|8|chocolate|light brown|4.75
Baskin|9|chocolate|dark brown|5.25
Baskin|10|strawberry|pink|5.25
Baskin|11|bubblegum|pink|4.75
Baskin|12|chocolate|dark brown|5.25
Baskin|13|strawberry|pink|3.55
Baskin|14|chocolate|light brown|4.75
Baskin|15|chocolate|dark brown|5.25
Baskin|16|strawberry|pink|5.25
Baskin|17|bubblegum|pink|4.75
Baskin|18|chocolate|dark brown|5.25
Baskin|19|strawberry|pink|3.55
Baskin|20|chocolate|light brown|4.75
Baskin|21|chocolate|dark brown|5.25
Baskin|22|strawberry|pink|5.25
Baskin|23|bubblegum|pink|4.75
Baskin|24|chocolate|dark brown|5.25
Baskin|25|strawberry|pink|3.55
Baskin|26|chocolate|light brown|4.75
Baskin|27|chocolate|dark brown|5.25
Baskin|28|strawberry|pink|5.25
Baskin|29|bubblegum|pink|4.75
Baskin|30|chocolate|dark brown|5.25

CREATE TABLE sales AS
    SELECT "Baskin" AS cashier, 1 AS TID UNION
    SELECT "Baskin", 2 UNION
    SELECT "Baskin", 3 UNION
    SELECT "Baskin", 4 UNION
    SELECT "Robin", 5 UNION
    SELECT "Robin", 6;

CREATE TABLE cones AS
    SELECT "Baskin" AS cashier, 1 AS TID UNION
    SELECT "Baskin", 2 UNION
    SELECT "Baskin", 3 UNION
    SELECT "Baskin", 4 UNION
    SELECT "Robin", 5 UNION
    SELECT "Robin", 6;

 Infer the result of the join operation:

```
**Inner Join**

```
select * from sales, cones where TID=ID;
```
SQL: using named tables - from

```sql
select "delicious" as Taste, Flavor, Color from cones
    where Flavor is "chocolate" union
select "other", Flavor, Color from cones
    where Flavor is not "chocolate";
```

```
sqlite> select "delicious" as Taste, Flavor, Color from cones where Flavor is "chocolate" union
[ ...> select "other", Flavor, Color from cones where Flavor is not "chocolate";
    Taste|Flavor|Color
delicious|chocolate|dark brown
delicious|chocolate|light brown
other|bubblegum|pink
other|strawberry|pink
sqlite>  
```
Queries within queries

• Any place that a table is named within a select statement, a table could be computed
  – As a sub-query

```sql
select TID from sales where Cashier is "Baskin";

select * from cones
  where ID in (select TID from sales where Cashier is "Baskin");

sqlite> select * from cones
 ...> where ID in (select TID from sales where Cashier is "Baskin");
ID|Flavor|Color|Price
1|strawberry|pink|3.55
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
```
Inserting new records (rows)

INSERT INTO table(column1, column2,...)
VALUES (value1, value2,...);

```sql
sqlite> insert into cones(ID, Flavor, Color, Price) values (7, "Vanila", "White", 3.95);
sqlite> select * from cones;
ID|Flavor|Color|Price
1|strawberry|pink|3.55
2|chocolate|light brown|4.75
3|chocolate|dark brown|5.25
4|strawberry|pink|5.25
5|bubblegum|pink|4.75
6|chocolate|dark brown|5.25
7|Vanila|White|3.95
sqlite> cones.append((7, "Vanila", "White", 3.95))

<table>
<thead>
<tr>
<th>ID</th>
<th>Flavor</th>
<th>Color</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>strawberry</td>
<td>pink</td>
<td>3.55</td>
</tr>
<tr>
<td>2</td>
<td>chocolate</td>
<td>light brown</td>
<td>4.75</td>
</tr>
<tr>
<td>3</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
<tr>
<td>4</td>
<td>strawberry</td>
<td>pink</td>
<td>5.25</td>
</tr>
<tr>
<td>5</td>
<td>bubblegum</td>
<td>pink</td>
<td>4.75</td>
</tr>
<tr>
<td>6</td>
<td>chocolate</td>
<td>dark brown</td>
<td>5.25</td>
</tr>
<tr>
<td>7</td>
<td>Vanila</td>
<td>White</td>
<td>3.95</td>
</tr>
</tbody>
</table>
```

- A database table is typically a shared, durable repository shared by multiple applications
Multiple clients of the database

- All of the inserts update the common repository
SQLite python API

In [64]: `import` sqlite3

In [65]: `icecream = sqlite3.connect('icecream.db')`

In [66]: `icecream.execute('SELECT * FROM cones;')`

Out[66]: `<sqlite3.Cursor at 0x111127960>`

In [67]: `icecream.execute('SELECT DISTINCT Flavor FROM cones;').fetchall()`

Out[67]: `[('strawberry',), ('chocolate',), ('bubblegum',)]`

In [68]: `icecream.execute('SELECT * FROM cones WHERE Flavor is "chocolate";').fetcha`

Out[68]: `[2, 'chocolate', 'light brown', 4.75),
(3, 'chocolate', 'dark brown', 5.25),
(6, 'chocolate', 'dark brown', 5.25)]`
class SQL_Table(Table):
    """ Extend Table class with methods to read/write a Table from/to a table in a SQLite3 database. """
    @classmethod
    def read(cls, filepath, table, verbose=False):
        """Create a SQL_Table by reading a table from a SQL database."""
        
        dbconn = sqlite3.connect(filepath,
                        detect_types=sqlite3.PARSE_COLNAMES)
        
        col_names = sqlcol_names(dbconn, table)
        rows = sqlexec(dbconn, 'SELECT * from %s;' % table, verbose).fetchall
        dbconn.close()
        return cls(col_names).with_rows(rows)
class SQL_Table(Table):
    ...
    def write(self, filepath, table, verbose=False, overwrite=True):
        """Write a Table into a SQL database as a SQL table."""

        dbconn = sqlite3.connect(filepath)
        
        # Create table and insert each row
        cols = build_list(self.labels)
        sqlexec(dbconn, "CREATE TABLE %s %s;" % (table, cols), verbose)
        for row in self.rows:
            sqlexec(dbconn, 'INSERT INTO %s VALUES %s;' % (table, tuple(row))
        dbconn.commit()
        dbconn.close()

@classmethod
def cast(cls, table):
    """Return a SQL_Table version of a Table."""
    return cls().with_columns(zip(table.labels, table.columns))
Summary – Part 1

SELECT <col spec> FROM <table spec> WHERE <cond spec> 
GROUP BY <group spec> ORDER BY <order spec>;

INSERT INTO table(column1, column2,...) 
VALUES (value1, value2,...);

CREATE TABLE name ( <columns> );

CREATE TABLE name AS <select statement> ;

DROP TABLE name ;
Summary

• SQL a declarative programming language on relational tables
  – largely familiar to you from data
  – create, select, where, order, group by, join

• Databases are accessed through Applications
  – e.g., all modern web apps have Database backend
  – Queries are issued through API
    » Be careful about app corrupting the database

• Data analytics tend to draw database into memory and operate on it as a data structure
  – e.g., Tables

• More in lab
Solutions for the Wandering Mind

1) Adding two n-bit integers, how many bits can the result have? Maximally n+1 bits (1 overflow bit).

2) Multiplying two n bit integers, how many bits can the result have?

\[ \log a \cdot b = \log a + \log b \rightarrow \log n \cdot n = 2 \cdot \log n. \] 2n bits.

Assume:

- a) Exceptions don’t exist
- b) We only reserve 8bit for an integer variable (0-255)

Questions:

1) What would be the result of an addition 255+255?

\[(255+255) \text{ modulo } 255 = 0. \] One entire summand is lost!

2) What would be the result of a multiplication 255*255?

\[(255 \times 255) \text{ modulo } 255 = 0. \] The error is: 64770!
3) Assume $l$ additions of 8bit integers into the same 8bit variable. Can you formulate the maximum error that can occur as a function of $l$?

Each addition can maximally lose the entire 8bits. The error accumulates. This is, each addition loses maximally 8bits. So it’s $l \times 8$ bits of loss. So the maximum error $255 \times l$.

For multiplication it’s $255^l$. This is, exponential error! (Also compare Lyapunov Exponent in physics)

This is taught in CS61C