SQL

Storing and querying structured data

By the end of this lecture, you should be able to:

- Given some data, say whether or not a relational database would be appropriate.
- Explain what structured data is
- Relate relational database concepts to OOP concepts
- Connect to a database through a client
- Use a SELECT query to get data from a single entity
  - Limit the set of rows using WHERE
  - Perform aggregate functions using GROUP BY
- Get information from multiple entities
  - Simple in-line combinations using subclauses
  - Complicated combinations using JOIN
- Compose SELECT, WHERE, GROUP BY, JOIN into single query

How to store data?

- The contents of Tolstoy’s War and Peace? In Russian? 紅樓夢 by 曹雪芹? In Chinese?
- The temperature at SFO every minute over a 100 year span?
- The location and size of every window currently open on your computer?
- The current inventory of the Trader Joe’s at 4th and Market?
- The bus and train (and cable car!) schedules for SF MUNI?

Data stores

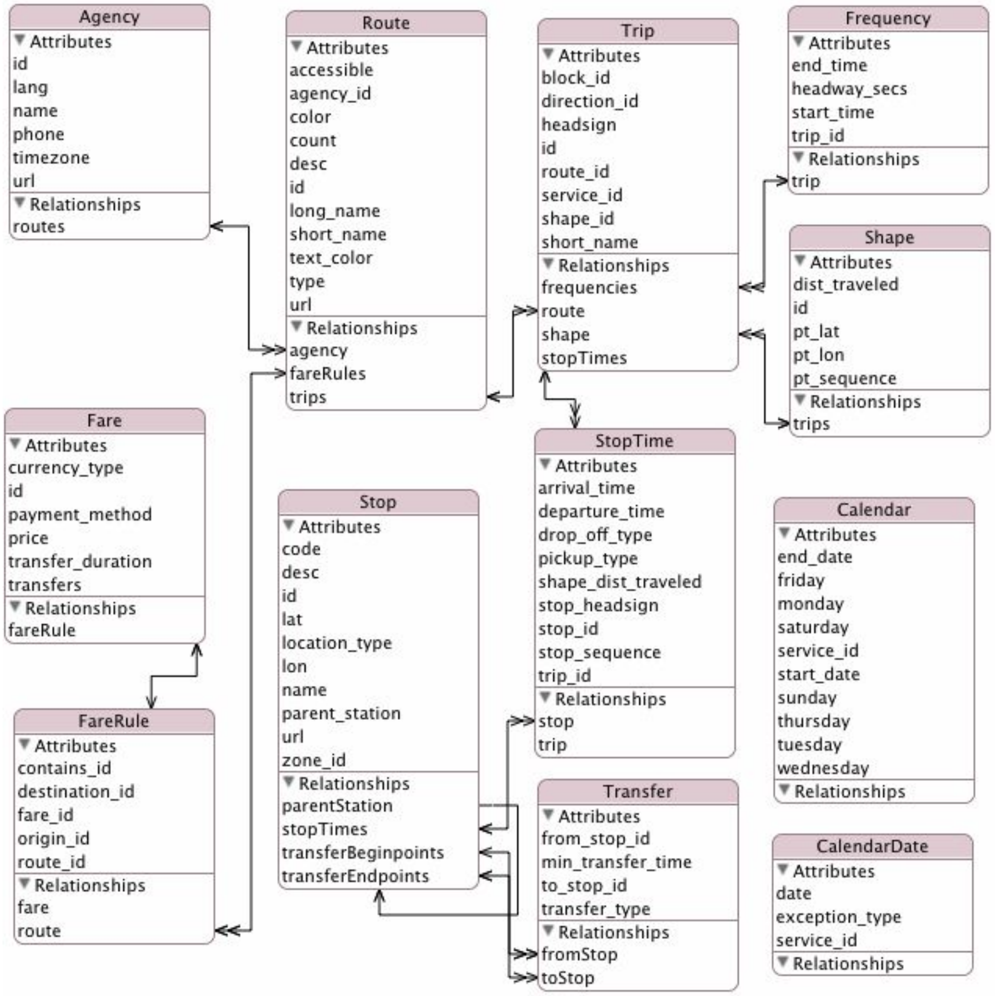
- Text files. (ASCII-encoded? UTF-8 encoded?)
- Arrays of floats. (Persisted? How?)
- In-memory data structure. (Persisted? Pickled?)
- CSV file? JSON file? How many CSV files?
- Database servers? What kind? **Relational**??

Relational Database Management System (RDBMS)

- Bigish
- Persistent
- Structured

Structured data

- Entities
  - Similar to classes
  - -methods
  - +flat namespace
  - +ids
- External references
  - Reference instances of other entities
    - E.g., “Product Count” references “Store”, “Product”
  - Can reference own entity
    - Person has attribute ‘mother’, which is a Person
- Data validity
  - Everyone has a mother.


Agency
Attributes
id
lang
name
phone
timezone
url
Relationships
routes
Route
Attributes
accessible
agency_id
color
count
desc
id
long_name
short_name
text_color
type
url
Relationships
agency
fareRules
trips
Trip
Attributes
block_id
direction_id
headsign
id
route_id
service_id
shape_id
short_name
Relationships
frequencies
route
shape
stopTimes
Frequency
Attributes
end_time
headway_secs
start_time
trip_id
Relationships
trip
Shape
Attributes
dist_traveled
id
pt_lat
pt_lon
pt_sequence
Relationships
trips
Fare
Attributes
currency_type
id
payment_method
price
transfer_duration
transfers
Relationships
fareRule
StopTime
Attributes
arrival_time
departure_time
drop_off_type
pickup_type
shape_dist_traveled
stop_headsign
stop_id
stop_sequence
trip_id
Relationships
stop
trip
Stop
Attributes
code
desc
id
lat
location_type
lon
name
parent_station
url
zone_id
Relationships
parentStation
stopTimes
transferBeginpoints
transferEndpoints
Calendar
Attributes
end_date
friday
monday
saturday
service_id
start_date
sunday
thursday
tuesday
wednesday
Relationships
FareRule
Attributes
contains_id
destination_id
fare_id
origin_id
route_id
Relationships
fare
route
Transfer
Attributes
from_stop_id
min_transfer_time
to_stop_id
transfer_type
Relationships
fromStop
toStop
CalendarDate
Attributes
date
exception_type
service_id
Relationships

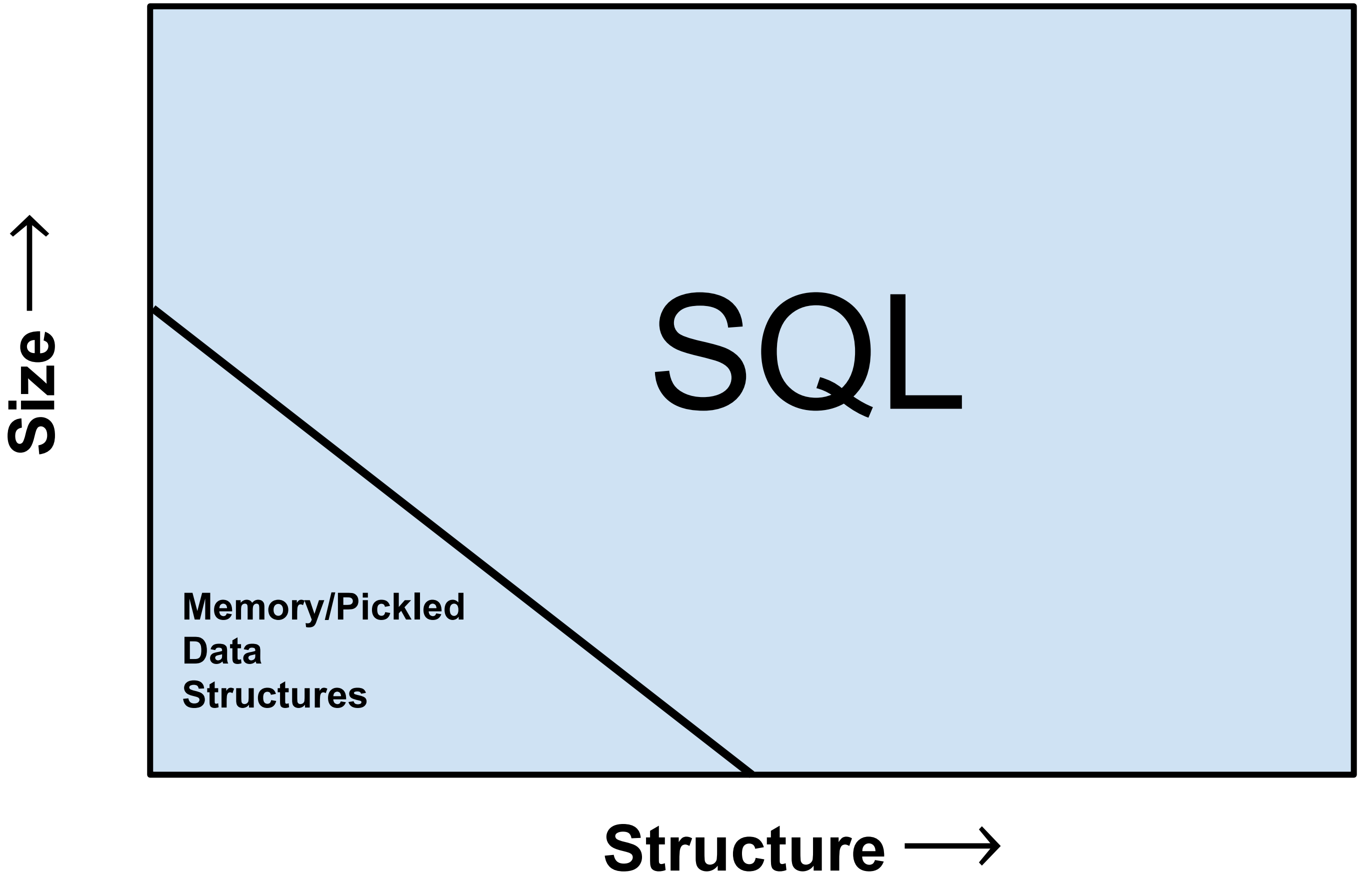
Connection to OOP concepts

- Class ↔ Table
- Instance ↔ Row
- Attribute ↔ Column
- Data Model ↔ Schema
- ??? ↔ Methods
- Joins ↔ ???
- Indexing ↔ ???

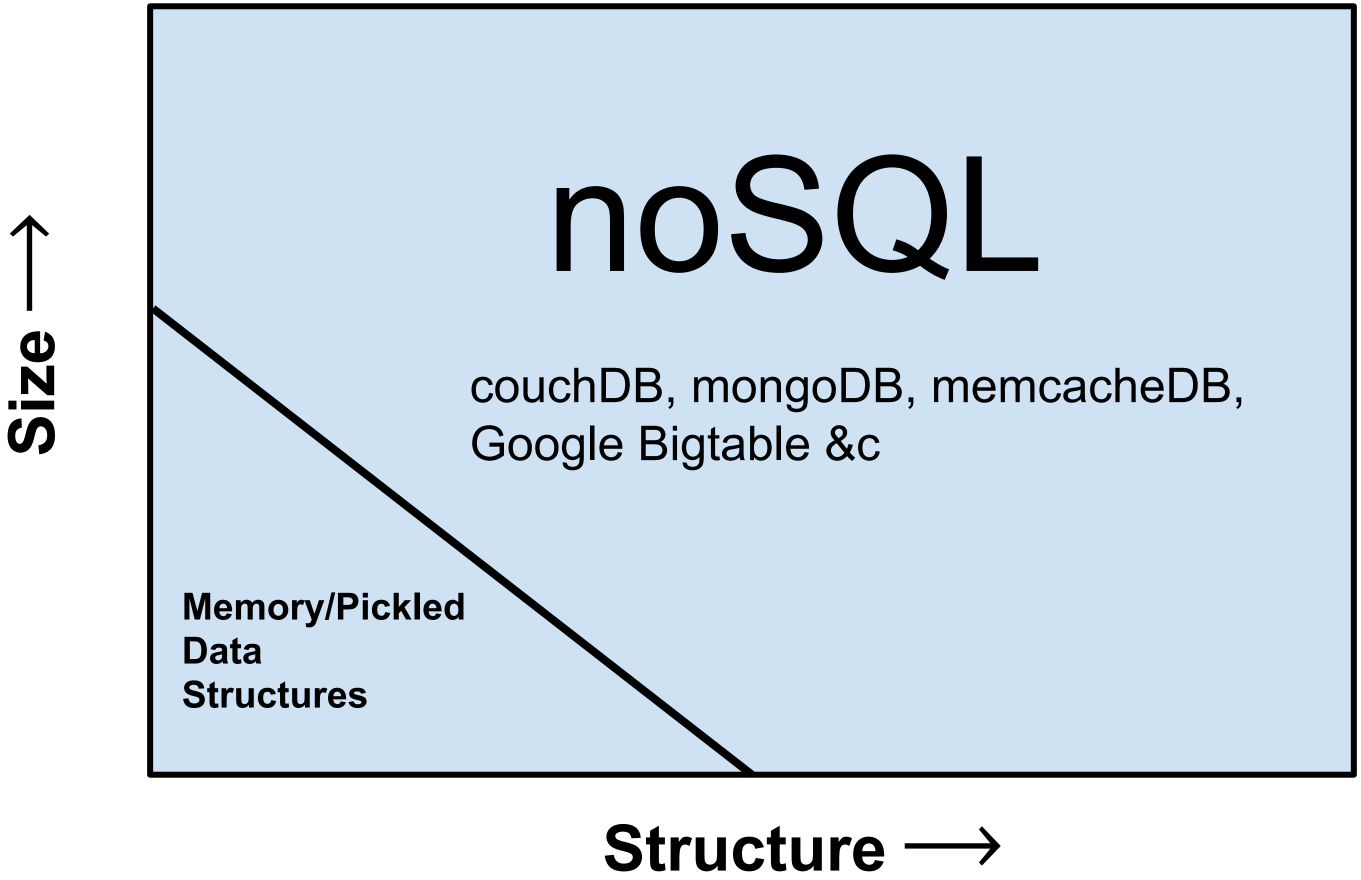
Warning: Object-Relational (ORM) Systems. Tempting, but irritating.

How do we all feel?

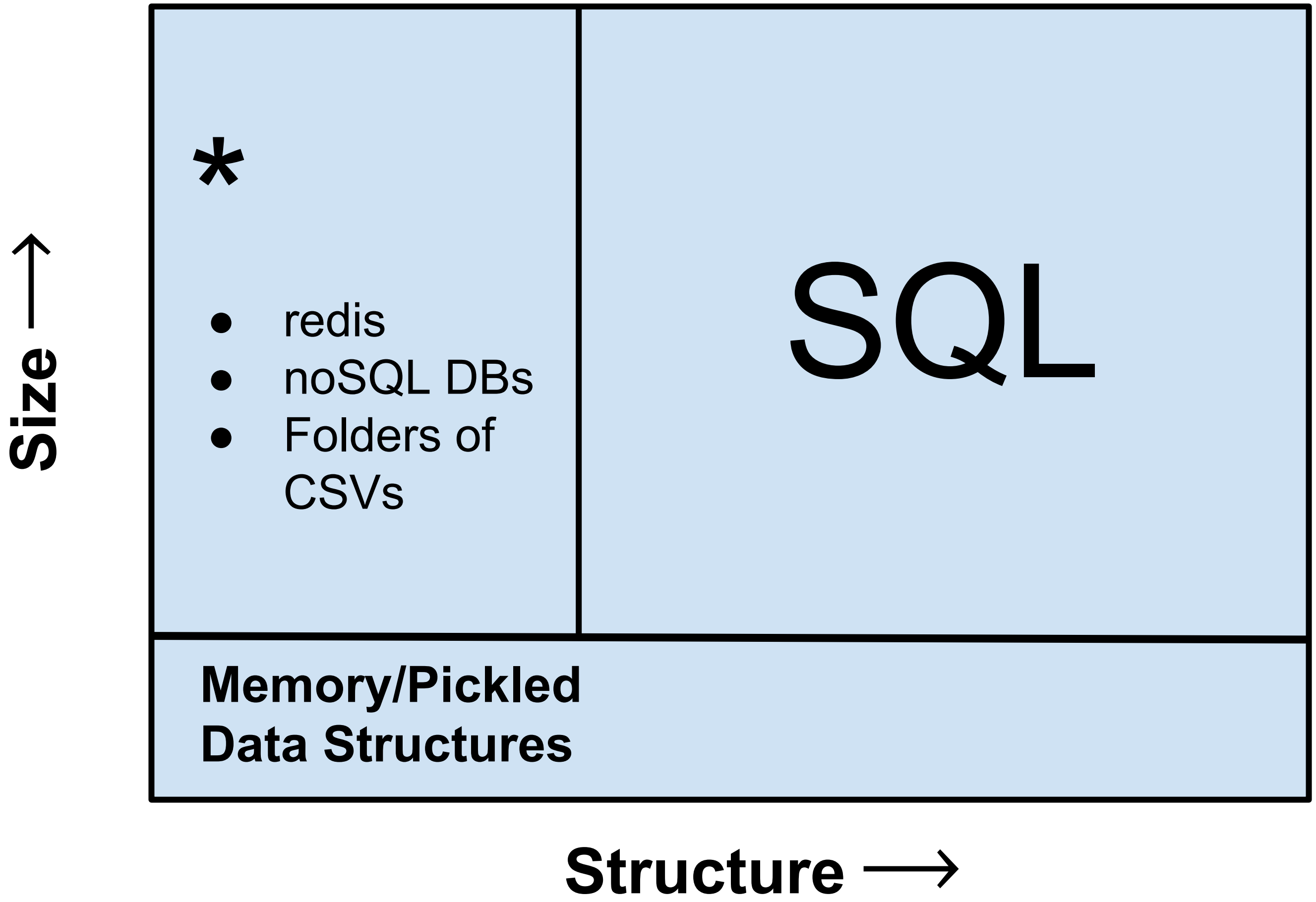
1985-2008 Memory expensive, CPUs slow, programming languages inconvenient



2008ish: The Dream



2008 - ?? : Memory cheap, CPUs fast, programming languages convenient



Other DBMS/SQL use cases:

- Legacy system support
- Tightly bound to web frameworks
- Concurrent access
- Referential integrity guarantees
- Data security
- **Querying & report generation**
    - “Remixable” data

Basic RDBMS/SQL Concepts

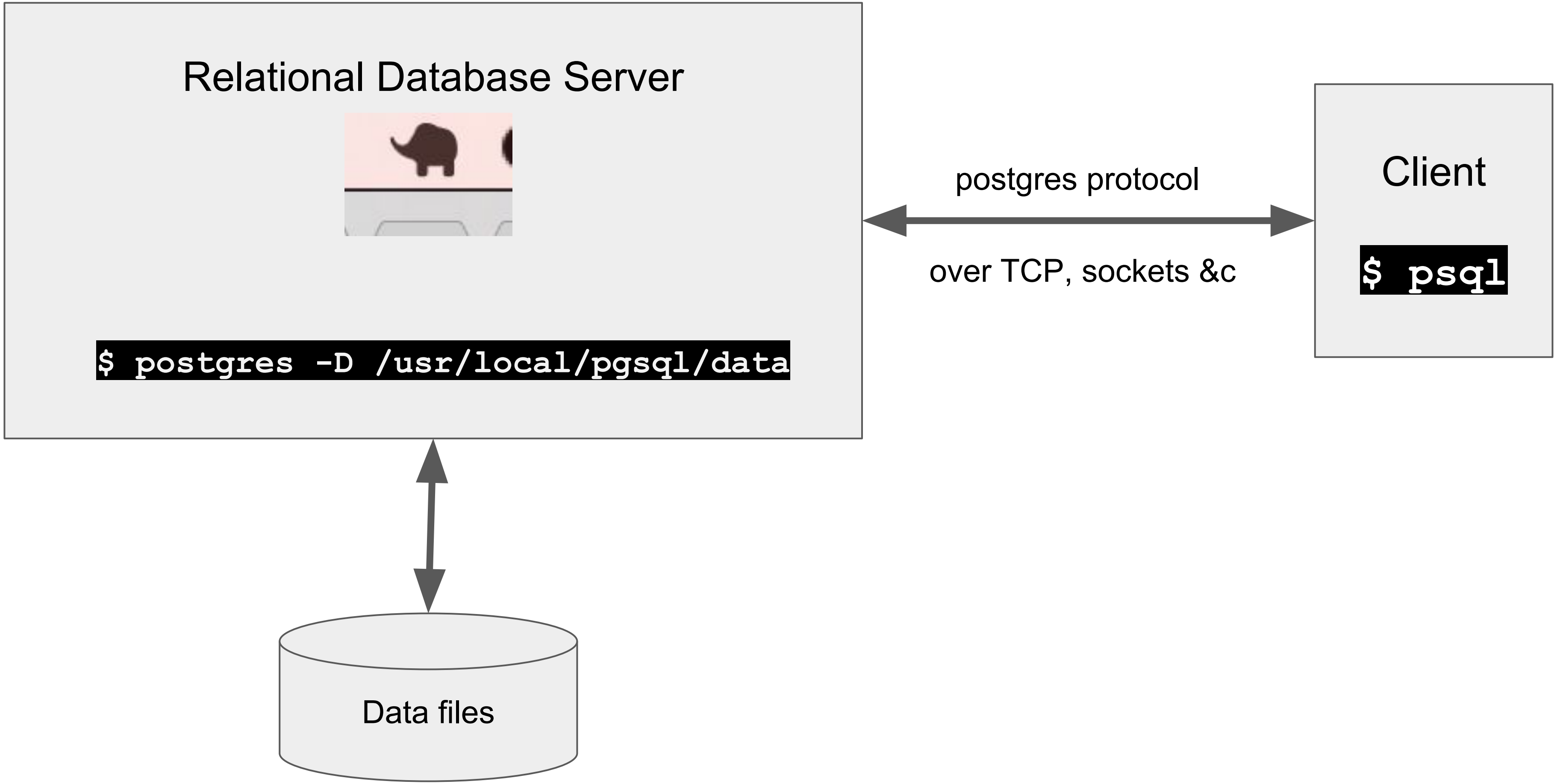
RDBMS Data Model

- **Schema** defines the structure of the data
- The **database** is composed of a number of user-defined **tables**
- Each **table** will have **columns** (aka fields) and **rows** (aka records)
- A column is of a given **data type**
- A row is an entry in a table with data for each column of that table

RDBMS and SQL (Structured Query Language)

- SQL is the language used to query relational databases
- **All RDBMS use SQL** and the syntax and keywords are the same for the most part, across systems
- **SQL is used to interact** with RDBMS, allowing you to create tables, alter tables, insert records, update records, delete records, and query records within and across tables.
- Even non-relational databases like **Hadoop** usually have a SQL-like interface available.

Client-Server Architecture


Relational Database Server
$ postgres -D /usr/local/pgsql/data
postgres protocol
over TCP, sockets &c
Client
$ psql
Data files

PostGres Basics

Ways to use psql in the shell/term:

`$ psql`	connects to postgres server
`$ psql -U [USERNAME]`	connects with given username
`$ psql [DBNAME]`	connects to a given database
`$ psql < script.sql`	reads file script.sql and send commands to psql

Try it live:

- *Open file sql/lecture_create.sql with a text editor*
- *Use it to create a “dsilecture” database on your psql server*

PostGres Basics

Useful psql commands at the prompt [link]:

`# \h`	SQL help
`# \?`	psql commands help
`# \l`	List all the tables in the database
`# \d`	Describe the table schema
`# \d db_name`	Describe tables for a specific db
`# \connect db_name`	Connects to a database

Try it live: Connect to “dsilecture” and describe schema of table “customer”

All together, now

```
$ cd ~/galvanize
$ mkdir sql-lecture
$ cd sql-lecture
$ wget
https://raw.githubusercontent.com/gSchool/DSI_Lectures/maste
r/sql/moses_marsh/sql/lecture_create.sql?token=AADLERkktd7xo
-j9nehgcPHqdS4HRyBqks5bocjfwA%3D%3D -O lecture_create.sql
$ psql < lecture_create.sql
$ psql dsilecture
```

SQL Syntax

All SQL queries have three main ingredient :

SELECT *What* data do you want?
FROM *Where* do you want to get the data from?
WHERE *Under what* conditions?

SQL is Declarative rather than Imperative. That is, you tell the machine what you want and it (database optimizer) decides how to do it

Advanced: You can use `EXPLAIN` to look at the how

Basic **SELECT**

```
SELECT * FROM customers;
```

Select columns

```
SELECT name, age, gender FROM customers;
```

Restrict rows with **WHERE**

```
SELECT * FROM customers
WHERE gender='M';
```

Limit number of responses

```
SELECT * FROM customers LIMIT 3
```

Basic aggregate functions

```
SELECT count(*) FROM customers;
```

Basic aggregate functions

```
SELECT min(age) AS min_age,
       max(age) AS max_age FROM customers;
```

Aggregating groups with `GROUP BY`

Find average customer age for each state

```
SELECT state, AVG(age) as avg_age
FROM customers
GROUP BY state;
```

How to query the “visits” table to get number of visits per customer?

What happens when we do this?

```
SELECT * FROM customers GROUP BY state;
```

Distinct, ordering

Select a distinct set

```
SELECT DISTINCT state FROM customers;
```

Ordering with ORDER BY

```
SELECT * FROM customers ORDER BY age;
```

To order by age in descending order:

```
SELECT * FROM customers ORDER BY age DESC;
```

How would we write a query to get the oldest customer?

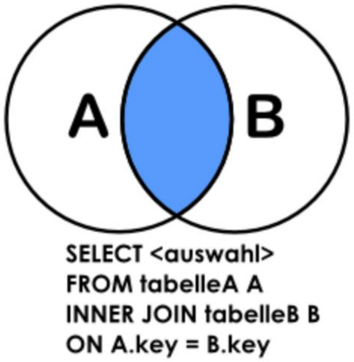
JOINS

The peanut butter and jelly of SQL

JOIN types

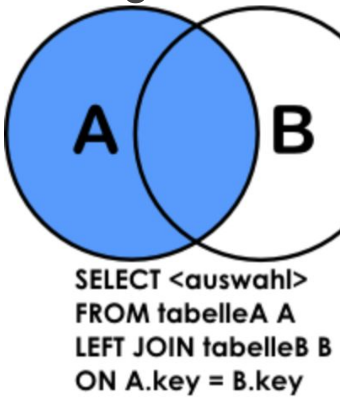
INNER JOIN

discards any entries
that do not have a match
between the tables
based on the given keys.



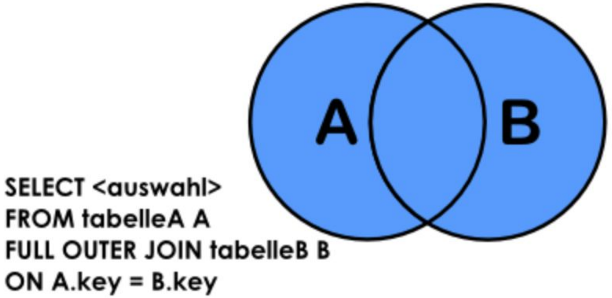
LEFT OUTER JOIN

keeps all entries
in the left table
regardless of
whether a match is found
in the right table



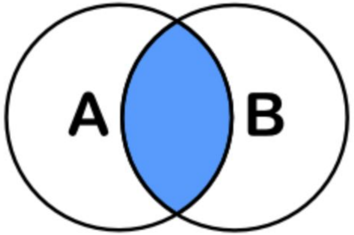
FULL OUTER JOIN

will keep the rows
of both tables
no matter what



copied from https://stackoverflow.com/questions/24257890/why-on-clause-used-on-full-outer-join

Inner Join




```
SELECT <auswahl>
FROM tabelleA A
INNER JOIN tabelleB B
ON A.key = B.key
```

first

id	name
1	Elliott
2	Mark
3	Moses
4	Brandon

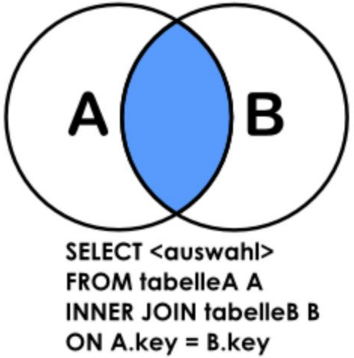
last

id	name
2	Llorente
3	Marsh
5	Engard
6	Van

```
SELECT id, first.name,
last.name
FROM first
INNER JOIN last
ON first.id=last.id
```

id	first	last

Inner Join



first

id	name
1	Elliott
2	Mark
3	Moses
4	Brandon

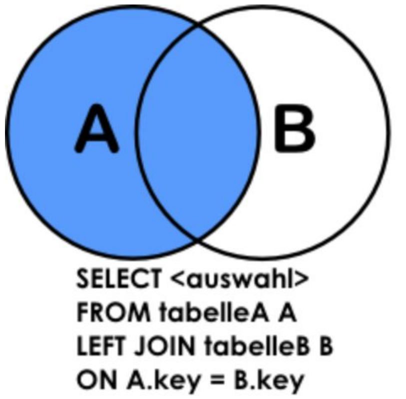
last

id	name
2	Llorente
3	Marsh
5	Engard
6	Van

```
SELECT id, first.name,
last.name
FROM first
INNER JOIN last
ON first.id=last.id
```

id	first	last
2	Mark	Llorente
3	Moses	Marsh

Left Outer Join



first

id	name
1	Elliott
2	Mark
3	Moses
4	Brandon

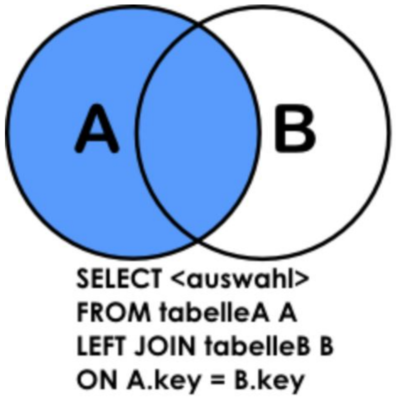
last

id	name
2	Llorente
3	Marsh
5	Engard
6	Van

```
SELECT id, first.name,
last.name
FROM first
LEFT JOIN last
ON first.id=last.id
```

id	first	last

Left Outer Join



first

id	name
1	Elliott
2	Mark
3	Moses
4	Brandon

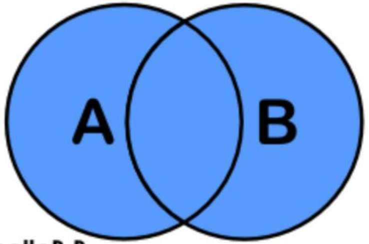
last

id	name
2	Llorente
3	Marsh
5	Engard
6	Van

```
SELECT id, first.name,
last.name
FROM first
LEFT JOIN last
ON first.id=last.id
```

id	first	last
1	Elliott	
2	Mark	Llorente
3	Moses	Marsh
4	Brandon	

Full Outer Join




```
SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key
```

first

id	name
1	Elliott
2	Mark
3	Moses
4	Brandon

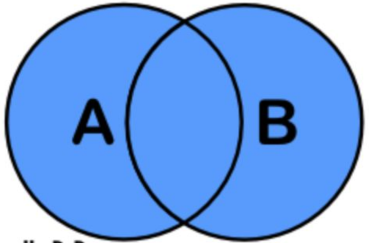
last

id	name
2	Llorente
3	Marsh
5	Engard
6	Van

id	first	last

```
SELECT id, first.name,
last.name
FROM first
FULL OUTER JOIN last
ON first.id=last.id
```

Full Outer Join



SELECT <auswahl>
FROM tabelleA A
FULL OUTER JOIN tabelleB B
ON A.key = B.key

first

id	name
1	Elliott
2	Mark
3	Moses
4	Brandon

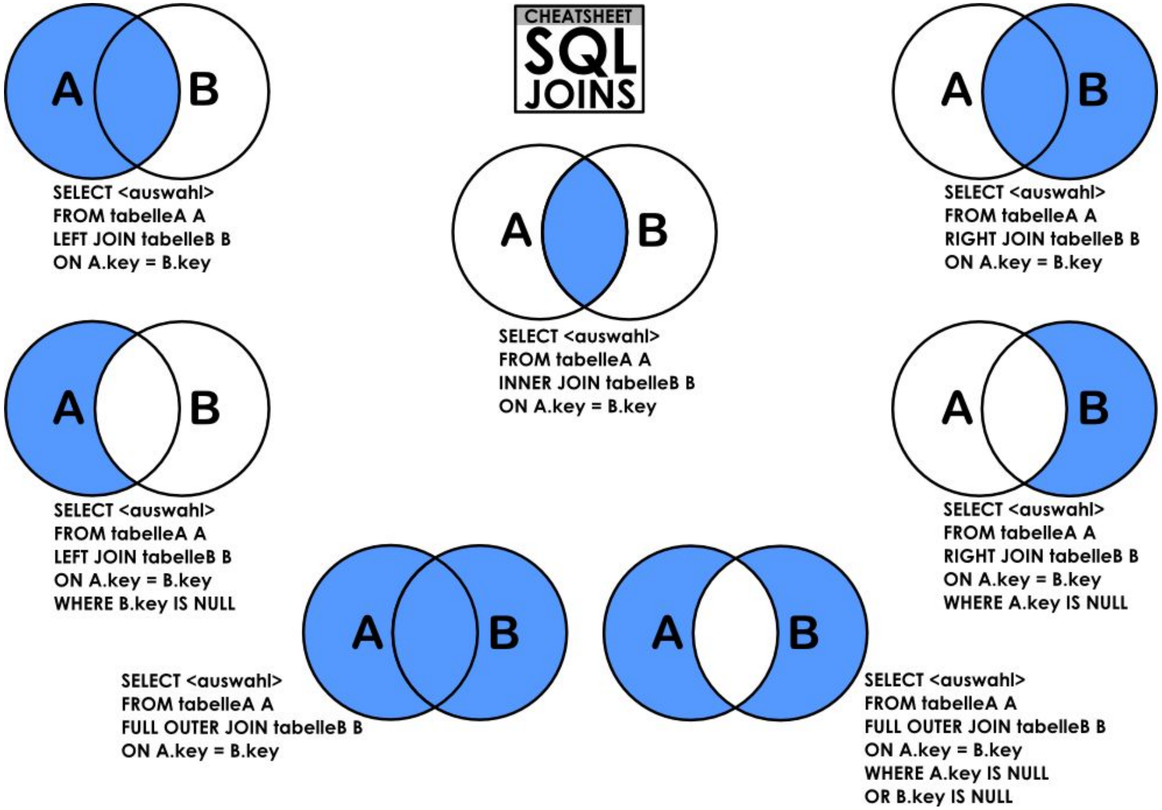
last

id	name
2	Llorente
3	Marsh
5	Engard
6	Van

```
SELECT id, first.name,
last.name
FROM first
FULL OUTER JOIN last
ON first.id=last.id
```

id	first	last
1	Elliott	
2	Mark	Llorente
3	Moses	Marsh
4	Brandon	
5		Engard
6		Van

Other types of joins



Composing SQL Queries

```sql
--- Return the customer_ids of all customers who visited in June ---
SELECT c.id, v.created_at
FROM customers as c
JOIN visits as v
ON c.id = v.customer_id
WHERE date_part('month', v.created_at) = 6;

--- LEFT JOIN: return all customers from the customers table regardless of presence in visits
SELECT
 c.id
, v.created_at
FROM customers as c
LEFT JOIN visits as v
ON c.id = v.customer_id
WHERE date_part('month', v.created_at) = 6;
```

Subqueries

In general, you can replace any table name with a SELECT statement.

```
SELECT ..... FROM (SELECT ....)
```

If a query returns a **single value**, you can treat it as such.

```
WHERE var1 = (SELECT ...)
```

If a query returns a **single column**, you can treat it sort of like a list/vector

```
WHERE var1 IN (SELECT ...)
```

Afternoon Lecture

More about SQL and RDBMSs

Given the following query, number what order the commands are executed:

```
SELECT a.userid, COUNT(*) AS recent_visits
FROM users AS a
LEFT JOIN visits AS b
ON a.userid = b.userid
WHERE b.dt > '2012-01-01'
GROUP BY a.userid
HAVING count(0) < 10
ORDER BY recent_visits;
```

Order of Evaluation of a SQL SELECT Statement

1. FROM + JOIN: first the product of all tables is formed

2. WHERE: the where clause filters rows that do not meet the search condition

3. GROUP BY + (COUNT, SUM, etc): the rows are grouped using the columns in the group by clause and the aggregation functions are applied on the grouping

4. HAVING: like the WHERE clause, but can be applied after aggregation

5. SELECT: the targeted list of columns are evaluated and returned

6. DISTINCT: duplicate rows are eliminated

7. ORDER BY: the resulting rows are sorted

Creating a table with a schema

table name

```
CREATE TABLE customers (
    id INTEGER PRIMARY KEY,
    name VARCHAR(50),
    age INTEGER,
    gender VARCHAR(1),
    city VARCHAR(255),
    state VARCHAR(2) );
```

columns / fields type

columns / fields name

Inserting values in a table

table name

records
and their values

```
INSERT INTO products (id, name, price) VALUES
   (1, 'soccer ball', 20.5),
   (2, 'iPod', 200),
   (3, 'headphones', 50);
```

SQL Queries for table creation / maintenance

Creating a table from query:

```
CREATE [TEMPORARY] TABLE table AS <SQL query>;
```

Inserting records in a table:

```
INSERT INTO table [(c1,c2,c3,…)] VALUES (v1,v2,v3,…);
```

Updating records:

```
UPDATE table SET c1=v1,c2=v2,… WHERE cX=vX;
```

Delete records:

```
DELETE FROM table WHERE cX=vX;
```

Change model (add, drop, modify columns):

```
ALTER TABLE table [DROP/ADD/ALTER] column [datatype];
```

Delete a table:

```
DROP TABLE table;
```

Designing a database with keys

Primary Key

A primary key is a special column of a table that uniquely identifies that entry.

A primary key is not always an integer; it could be a combination of columns, hash, timestamp..etc.,

Foreign Keys

Foreign Keys are columns that reference some other entry in the database.

Database Normalization

- Minimizes Redundancy, for example:
    - Details about a user(address, age) are only stored once (in a users table)
    - Any other table (eg. purchases) where this data might be relevant, only references the user_id
    - Choose Normalized or Denormalized Schemas based on the use case:
        - Heavy reporting (Data Warehouse)
        - Transactional Systems (Ordering System)

Advanced SQL

- "self join": JOIN a table with itself: http://www.w3resource.com/sql/joins/perform-a-self-join.php
- CROSS JOIN: join each row in table A with every row in table B: http://www.w3resource.com/sql/joins/cross-join.php
- window functions: https://www.postgresql.org/docs/9.1/static/tutorial-window.html
- COALESCE: often used to turn NULL values into non-null values: https://www.postgresql.org/docs/9.5/static/functions-conditional.html
- EXCEPT: return rows from one SELECT statement that are NOT returned by a second SELECT statement: https://www.tutorialspoint.com/sql/sql-except-clause.htm