Relational Data Processing

Everything Data
CompSci 216 Spring 2019
Structure is good

• More structure ➔ easier, more powerful analysis
• Remember OpenRefine

Spreadsheets allow sorting, filtering, grouping, counting, summing…
But … consider the following problem

Find the longest-serving current members of the Congress

- A member may serve multiple terms
- So data has a nested structure
Spreadsheet chokes... on this and other more complex structures
How do we structure data now?

One table ➔ multiple tables
Persons & their roles

**persons**

- **Birthday**
- **Gender**
- Start/end of a term
- Party
- State
- House/senate?
- Person ID
- Name

One row per person

**person_roles**

- One row for each term served by a person
Relational data model

How is data structured/constrained?
• Organize data in tables (AKA relations)
  – Each table has a list of (typed) columns
    • Data is stored as rows
    • Each row has a value for every column
• Declare structure + constraints as schema

How is data queried/updated?
• A “declarative” language called SQL
  – Say what result you want, not how to compute it
CREATE TABLE persons (  id CHAR(10) NOT NULL PRIMARY KEY,  id_govtrack INTEGER NOT NULL UNIQUE,  id_lis CHAR(4) UNIQUE,  first_name VARCHAR(50) NOT NULL,  middle_name VARCHAR(50),  last_name VARCHAR(50) NOT NULL,  birthday DATE,  gender CHAR(1)  CHECK (gender IS NULL OR gender IN ('F', 'M')) );
CREATE TABLE person_roles (  
    person_id CHAR(10) NOT NULL REFERENCES persons(id),  
    type CHAR(3) NOT NULL CHECK (type IN ('rep', 'sen')),  
    start_date DATE NOT NULL,  
    end_date DATE NOT NULL,  
    state CHAR(2) NOT NULL REFERENCES states(id),  
    district INTEGER CHECK ((type = 'rep' AND district IS NOT NULL) OR  
                            (type = 'sen' AND district IS NULL)),  
    party VARCHAR(20)  
);  

Foreign-key constraint: every person_id value must be some id value found in persons; i.e., no “dangling” references
## Example data

### persons

<table>
<thead>
<tr>
<th>id</th>
<th>id_govtrack</th>
<th>id_li</th>
<th>first_name</th>
<th>middle_name</th>
<th>last_name</th>
<th>birthday</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>D00062</td>
<td>412533</td>
<td></td>
<td>Tammy</td>
<td></td>
<td>Duckworth</td>
<td>1968-03-12</td>
<td>F</td>
</tr>
<tr>
<td>T000476</td>
<td>412668</td>
<td>S384</td>
<td>Thom</td>
<td></td>
<td>Tillis</td>
<td>1960-08-30</td>
<td>M</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### person_roles

<table>
<thead>
<tr>
<th>person_id</th>
<th>type</th>
<th>start_date</th>
<th>end_date</th>
<th>state</th>
<th>distric</th>
<th>party</th>
</tr>
</thead>
<tbody>
<tr>
<td>D00062</td>
<td>rep</td>
<td>2013-01-03</td>
<td>2015-01-03</td>
<td>IL</td>
<td>8</td>
<td>Democrat</td>
</tr>
<tr>
<td>D00062</td>
<td>rep</td>
<td>2015-01-03</td>
<td>2017-01-03</td>
<td>IL</td>
<td>8</td>
<td>Democrat</td>
</tr>
<tr>
<td>T000476</td>
<td>sen</td>
<td>2015-01-06</td>
<td>2021-01-03</td>
<td>NC</td>
<td></td>
<td>Republican</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
For simplicity...

Assume we have this “table”:

- Columns in the primary key are underlined

\texttt{cur\_members(id, first\_name, last\_name, gender, birthday, type, party, state)}

\textit{We will come back to how to create this “table” later}
Show me a table... sorted

• List all current members of the Congress

```sql
SELECT * -- A shorthand for "all columns"
FROM cur_members; -- Marks the end of the query
```

• **Sorting** options
  - ... `ORDER BY birthday;` (Default is ascending order)
  - ... `ORDER BY birthday DESC;`
  - ... `ORDER BY type, party;`

`cur_members(id, first_name, last_name, gender, birthday, type, party, state)`
Picking columns

- AKA *projection*

```sql
SELECT id, first_name, last_name, state, type,
  (date_part('year', current_date) -
   date_part('year', birthday)) AS age
FROM cur_members
ORDER BY age;
```

You can compute a new column to output ... and give it a name

`cur_members(id, first_name, last_name, gender, birthday, type, party, state)`
Picking rows

- AKA *filtering* or *selection*

```sql
SELECT *
FROM cur_members
WHERE type = 'sen' AND birthday >= '1950-01-01';
```

- Comparison, not assignment
- Strings are enclosed by single quotes
- ... AND ... OR ...
- NOT (...)

`cur_members(id, first_name, last_name, gender, birthday, type, party, state)`
Grouping and aggregating rows

SELECT party, COUNT(*)
FROM cur_members
GROUP BY party;

Put members of the same party in one group

Count the size of each group

cur_members(id, first_name, last_name, gender, birthday, type, party, state)
More grouping/aggregation

```
SELECT party, gender, COUNT(*),
    AVG(date_part('year', current_date) - date_part('year', birthday))
FROM cur_members
GROUP BY party, gender;
```

Rows now must match on both columns to be in the same group.

Again, one output row per group.

Other aggregation functions include
`SUM`, `MAX`, `MIN`

`cur_members(id, first_name, last_name, gender, birthday, type, party, state)"
Joining tables

• How did we get cur_members(id, first_name, last_name, gender, birthday, type, party, state)?

• Need to “join” tables together

persons(id, id_govtrack, id_lis, first_name, last_name, birthday, gender)
person_roles(person_id, type, start_date, end_date, state, district, party)
A SQL Query walks into a bar. In the corner of the bar are two tables. The Query walks up to the tables and asks,

“Mind if I join you?”
Join = pairing “related” rows

```
<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>D000622</td>
<td>Tammy</td>
<td>Duckworth</td>
<td>...</td>
</tr>
<tr>
<td>T000476</td>
<td>Thom</td>
<td>Tillis</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>person_id</th>
<th>start_date</th>
<th>end_date</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>D000622</td>
<td>2013-01-03</td>
<td>2015-01-03</td>
<td>...</td>
</tr>
<tr>
<td>D000622</td>
<td>2015-01-03</td>
<td>2017-01-03</td>
<td>...</td>
</tr>
<tr>
<td>T000476</td>
<td>2015-01-06</td>
<td>2021-01-03</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```

“Join condition” is persons.id = person_roles.person_id

Output table:

```
<table>
<thead>
<tr>
<th>id</th>
<th>first_name</th>
<th>last_name</th>
<th>person_id</th>
<th>start_date</th>
<th>end_date</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>D000622</td>
<td>Tammy</td>
<td>Duckworth</td>
<td>D000622</td>
<td>2013-01-03</td>
<td>2015-01-03</td>
<td>...</td>
</tr>
<tr>
<td>D000622</td>
<td>Tammy</td>
<td>Duckworth</td>
<td>D000622</td>
<td>2015-01-03</td>
<td>2017-01-03</td>
<td>...</td>
</tr>
<tr>
<td>T000476</td>
<td>Thom</td>
<td>Tillis</td>
<td>T000476</td>
<td>2015-01-06</td>
<td>2021-01-03</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
cur_members in SQL

CREATE VIEW cur_members AS
SELECT p.id, p.first_name, p.last_name, p.gender, p.birthday,
    r.type, r.party, r.state
FROM persons p, person_roles r
WHERE p.id = r.person_id
    AND r.start_date <= current_date
    AND current_date <= r.end_date;

Want to “save” your output for later querying?
Create a view—a “virtual” table

List tables to be joined
Join condition
Selection conditions local to r

Make p an alias for persons—think of it as a variable iterating through persons rows
Putting it together

**SELECT** *columns or expressions*
  (or for each group of them if query has grouping/aggregation)

**FROM** *tables*
  1. Generate all combinations of rows, one from each table; each combination forms a “wide row”

**WHERE** *conditions*
  2. Filter—keep only “wide rows” satisfying *conditions*

**GROUP BY** *columns*
  3. Group—“wide rows” with matching values for *columns* go into the same group

**ORDER BY** *output columns;*
  4. Compute one output row for each “wide row”
  5. Sort the output rows
Subqueries and **LIMIT**

- Find the ten longest serving members

SELECT id, first_name, last_name, birthday, 
(SELECT SUM(end_date - start_date) 
FROM person_roles r 
WHERE r.person_id = p.id) AS duration 
FROM persons p 
ORDER BY duration DESC LIMIT 10;

Pretend that for every \( p \) in persons we examine, we run the subquery with \( p \)'s id value

Just give me the first 10 rows
One more example

How does he vote?
Say, comparing with

- Pelosi (D, CA), minority leader

Rep. (& Prof.) David Price
(D, NC 4th District)
Expanded schema

**persons**
- id
- ...
- first_name
- last_name
- ...

**person_roles**
- person_id
- type
- start_date
- end_date
- ...

**votes**
- vote_id
- category
- chamber
- session
- date
- ...

**person_votes**
- vote_id
- person_id
- vote
Here we go…

WITH creates a temporary view for the query that follows

WITH votes_compare(vote_id, vote1, vote2) AS
(SELECT v1.vote_id, v1.vote, v2.vote
 FROM votes v, persons p1, persons p2, person_votes v1, person_votes v2
 WHERE v.chamber = 'h' AND (v.session = 2015 or v.session = 2016)
     AND p1.last_name = 'Price' AND p2.last_name = 'Pelosi'
     AND v1.person_id = p1.id AND v2.person_id = p2.id
     AND v1.vote_id = v2.vote_id AND v.id = v1.vote_id)

SELECT COUNT(*) AS agree,
    (SELECT COUNT(*) FROM votes_compare) AS total,
    COUNT(*)*100.00 / (SELECT COUNT(*) FROM votes_compare) AS percent
FROM votes_compare
WHERE vote1 = vote2;
... and the answer is:

| agree | total | percent             |
|-------+-------+---------------------|
| 1411  | 2649  | 53.2653831634579086 |

What's going on? Isn't Price a Democrat?

| agree | total | percent             |
|-------+-------+---------------------|
| 1100  | 1325  | 83.0188679245283019 |

It's your job to clear Prof. Price's name!

Dear chain-mailer: This is an exercise teaching students how to catch subtle errors in SQL queries—the numbers here are WRONG and just don’t grab them mindlessly.