Databases & SQL for Social Data

Jared Edgerton

Note on data. This week uses **synthetic** (simulated) campaign finance-style data generated in the course script. The goal is to practice database thinking and SQL—not to draw substantive inferences about real campaigns.

Conceptual Questions

Please write three to ten sentence explanations for each of the following questions. **You are only required to answer ONE of the two questions below.**

1. Explain what a **relational schema** is and why it is useful for social data. In your answer, define **primary keys** and **foreign keys**, and explain how they reduce duplication and enable joins. Use the (candidate, contributor, contribution) setting from this week's coding lab as your concrete example.
2. Explain what a database **index** does and why it can make queries faster. What are two trade-offs of adding indexes (e.g., disk usage, slower inserts/updates, maintenance)? Give one example of a query pattern (filter, join, group-by) that is likely to benefit from an index in the lab dataset.

Applied Exercises

Use the code in the week's code tutorial and the lecture slides to answer the following questions.

3. **Build the database + inspect the schema (synthetic data).** Using the provided @MastersThesis•, author = •, title = •, school = •, year = •, OPTkey = •, OPTtype = •, OPTaddress = •, OPTmonth = •, OPTnote = •, OPTannote = • script, create the SQLite database (`campaign_finance.db`) and load the synthetic tables: `candidates`, `contributors`, and `contributions`.
   - Report the row counts in each table using `SELECT COUNT(*)`.
   - Show the schema for each table (e.g., `PRAGMA table_info(candidates)` and similarly for the other two tables).
   - Briefly explain (2–4 sentences) how `contributor_id` and `candidate_id` connect the tables.
4. **Joins + aggregation (write and run your own SQL).** Write a SQL query (and run it through R or Python) that uses at least **one join** and at least **one aggregation**:
   - Required: join `contributions` to `candidates` and compute total contributions by `party`.
   - Required: restrict to contributions with `amount > 1000`.
   - Output: a clean table with columns `party`, `total_amount`, and `num_contributions`.
   - Visualization: make a simple bar plot of `total_amount` by party.

5. **Indexes + query plan (evidence of optimization).** Using SQL statements, do the following:
   - Verify which indexes exist on `contributions` (e.g., query `sqlite_master`).
   - Choose one query that filters by `candidate_id` *or* `date` *or* `amount`. Run `EXPLAIN QUERY PLAN` for that query.
   - In 4–6 sentences, interpret the query plan: does SQLite report using an index? If not, what index might help and why?
6. **Challenge Question (Optional — if you finish early):** Run the same join-and-aggregate query in **DuckDB** (from R or Python) and compare it to SQLite.
   - Repeat the query in DuckDB (you may load the tables from CSV or connect to the SQLite database, depending on what you set up in lab).
   - Provide one piece of evidence about performance (e.g., a timing using `system.time()` in R or timing in Python).
   - In 3–6 sentences, explain what factors might drive differences (data size, indexing, execution engine, I/O, query planner).