Relational Databases and SQL, Part I
Statistics 650/750
Week 7 Thursday

Christopher Genovese and Alex Reinhart

11 Oct 2018

1 Announcements

- Notes and ClassFiles in documents repository (`Lectures` and `ClassFiles/week7`, respectively)
- See Instructions page for details on connecting to SQL server

2 Where do you store your data?

You have a data set that you want to use in your work. How do you store it so that you can use it most effectively?

A few common scenarios:

- **Keep data in an ASCII file** (e.g., CSV file)

  Pros Easy to read, easy to edit, easy to archive and transfer

  Cons No checking of data, low data density, hard to search/query, must keep in sync with version used in software, requires separate file for documentation that must be kept in sync

  Questions
  1. If you read the data into an R data frame and change it, which is the authoritative copy of the data?
  2. If you make a mistake editing the file, what happens?
  3. What is needed to put a comma in a CSV field?

- **Keep data in an encoded file**

  Pros High data density (i.e., compact)

Cons Same problems as for ASCII file but more intense because you need a program to read the data

- **Keep data in an R Data Frame (or analogous data structure)**

  Pros Easy to use from a program, can attach metadata (e.g., documentation)

  Cons Not persistent, requires translation to use from another platform

- **Keep data in memory**

  Pros Very fast access, flexible structuring of the data

  Cons Not persistent, not accessible in parallel

A **database** is a structure for organizing information that can be *efficiently* accessed, updated, and managed – at scale. It is designed to address the many problems of these more informal approaches.

There are several different models for constructing databases, many of them new and optimized for large data sets in some way. But relational databases (RDBs) have been the dominant model since the 1970s and still remains popular and important. So we will discuss them first.

Relational databases tend to be *design-first* systems. First, you specify a *schema* for your data, and then you enter data that conforms to that schema. A properly designed schema can provide very flexible and powerful queries.

Relational Databases (to be called RDBs or databases for the next few classes) are commonly manipulated and queried using a (mostly) standardized language called **SQL**, which stands for Structured Query Language.

We will be using a powerful, open-source RDB system called **postgreSQL** (aka. "postgres"). It is fast, flexible, reliable, and can handle large data sets. It is highly compliant to the ANSI-standard for SQL and has some nice extensions. And it has been used successfully for many production systems over many years.

3 How do you use your data?

With rich data sets, we have the potential to answer many different kinds of questions. As we work with the data, we construct new view of the data, new summaries, new statistics and then pose new questions.

This involves cycles of **update** and **query**. The basic operations on our data involve adding variables, changing values, creating summaries, selecting data that meet certain criteria, adding or removing cases that meet those criteria, and establishing relationships between different entities in our data.

These fundamental operations are supported in many frameworks and platforms, often with different syntax. For example, in R, the `dplyr` package gives a set of operations for updating and querying a data frame. But the ideas and the cycle of operation is similar.

We will look at SQL, a key language for expressing these operations, in detail, but keep in mind that the concepts are quite general and broadly applicable.

4 Plan

4.1 Today

- Database Concepts
- Practical Introduction to Postgres
- Making Tables
- CRUD Operations

4.2 Next Time

- Schema Design
- Joins and Foreign Keys
- Using RDBs Programatically from Your Favorite Programming Language

4.3 Appendix For Reference

- A Few Advanced Maneuvers
- A Quick View of Other Database Models

5 A Few Database Concepts

5.1 ACID Guarantees

An RDB stores our data, and we read and operate on that data through requests sent to the database. These requests are formally called **transactions**.

Modern RDBs may receive many transactions at once, often operating on the same pieces of data. Particular care is needed to ensure that transactions are performed reliably and consistently.

For example, consider what would happen in the following cases:

- A transaction for a commercial payment is transfering money from your bank account and to another account. But the process ends after the money is deduced from one account but before adding it to the other.
- A similar transaction completes *just* before the power goes out in the server room
- A similar transaction completes even though you don't have enough money in your account to make the payment.

These are all boundary cases, but they can happen. And if they do, the viability of the entire system can be compromised.

So, RDBs are designed to make several strong guarantees about their performance, the so-called ACID guarantees:

- **Atomic**

  A transaction either succeeds entirely or fails leaving the database unchanged.

- **Consistent**

  Atransaction must change the database in a way that maintains all defined rules and constraints.

- **Isolated**

  Concurrent execution of transactions results in a transformation that would be obtained if the transactions were executed serially.

- **Durable**

  Once a transaction is committed, it remains so even in the face of crashes, power loss, and other errors.

This is another advantage of RDBs over ad hoc data storage.

5.2 Data Types

The **type** of a piece of data describes the set of possible values that data can have and the operations that can apply to it.

In an RDB, we specify the type of each data attribute in advance. Postgres, for instance, supports a wide variety of data types, including:

- Numeric Types, such as integers, fixed-precision floating point numbers, arbitrary precision real numbers, and auto-incrementing integer (`serial`).
- Text, including fixed-length and arbitrary character strings.
- Monetary values
- Date and Time Stamps
- Boolean values
- Geometric types, such as points, lines, shapes
- Elements in sets
- JSON structures

See the Postgres documentation on "Data Types" for details and for more examples.

5.3 Tables (Relations, Schemas, Entities)

The basic unit of data storage in an RDB is the **table**. Tables are also sometimes called *relations*, *schemas*, and *entities* in an RDB context.

A table is defined by its *attributes*, or columns, each of which has a **name** and a **type**.

Each *row* of a table defines a mapping from attribute names to values.

id	time	persona	element	latency	score	feedback
17	2015-07-11 09:42:11	3271	97863	329.4	240	Consider…
18	2015-07-11 09:48:37	3271	97864	411.9	1000	
19	2015-07-08 11:22:01	499	104749	678.2	750	The mean is…
22	2015-07-30 08:44:22	6742	7623	599.7	800	Try to think of…
24	2015-08-04 23:56:33	1837	424933	421.3	0	Please select…
32	2015-07-11 10:11:07	499	97863	702.1	820	What does the…
99	2015-07-22 16:11:27	24	88213	443.0	1000	

What are the attribute names and types for this table?

5.4 Unique, Primary, and Foreign Keys

It is valuable (even necessary) in practice for each row of a database table to be distinct. To that end, it is common to define a **unique key** – one or more attributes whose collective values uniquely identify every row.

In the Events table above, `id` is a unique key consisting of a single attribute.

There may be more than one unique key in a table, some resulting from the joint values of several attributes. One of these keys is usually chosen as the **primary key** – the key that is used in queries and in other tables to identify particular rows.

In the Events table above, `id` is also the primary key for the table. In practice, the primary key is often an auto-incrementing, or `serial`, integer like this.

When a table's primary key is used as an attribute in another table, it acts as a link to a row in the first table. A key used in this way is called a **foreign key**. Columns that store foreign keys are used for linking and cross-referencing tables efficiently.

In the Events table above, the `persona` and `element` attributes are foreign keys, referencing other tables, which I have not shown you.

5.5 Relationships Between Tables

We can think of tables as representing some entity that we are modeling in our problem. For example, each row of `Events` represents a single "event" of some sort; each persona in the `Personae` table represents a single student in a single class (in a specified term).

We link tables to define **relationships among entities**.

For example, each `persona` is linked to many events, while each event has a single associated `persona` and `element`.

A good *design* of the database tables can make it more efficient to query these relationships.

6 Introducing SQL and Postgres

6.1 Getting Started

Use instruction page to connect to the SQL server. Invoke `psql` at the shell prompt to start the PostgreSQL REPL.

Mac users with homebrew, might just want to install postgres directly with

```
brew install postgresql
pg_ctl -D /usr/local/var/postgres start
createdb NAME
```

where NAME is your username (the word after `/Users` in your home directory path).

6.1.1 Getting Help

Type '\?' at the prompt to get a list of meta-commands (these are system, not SQL commands).

A few of these are quite common:

- `'\h'` provides help on an SQL command or lists available commands
- `'\d'` list or describe tables, views, and sequences
- `'\l'` lists databases
- `'\c'` connect to a different database
- `'\i'` read input from a file (like source)
- `'\o'` send query output to a file or pipoe
- `'\!'` execute a shell command
- `'\cd'` change directory
- `'\q'` quit psql

6.1.2 Commands and Files

Update the `documents` repository from github. There are several files in `ClassFiles/week7` that you should copy into a working directory for this class:

- `instructions`
- `events.csv`
- `events.sql`
- `commands.sql`

The last of these is a text file containing sql commands that you can copy and paste into the prompt to save typing. Of course, typing the commands is fine too and is not a bad way to get a feel for how the commands work.

6.2 Entering SQL Statements

SQL consists of a sequence of *statements*.

Each statement is built around a specific command, with a variety of modifiers and optional clauses.

SQL statements can span several lines, and all SQL statements end in a semi-colon (;).

Keep in mind: strings are delimited by single quotes 'like this', *not* double quotes "like this".

SQL comments are lines starting with `--`.

To get help:

- You can get brief help on any SQL command with `\h <command>`.
- You can get detailed and helpful information on any aspect of postgres through the online documentation.
- The stat server is running version 9.2, that that will be updated if needed.

6.3 A Simple Example

Try the following (or copy it from the given file).

```
create table products (
       product_id SERIAL PRIMARY KEY,
       name text,
       price numeric CHECK (price > 0),
       sale_price numeric CHECK (sale_price > 0),
       CHECK (price > sale_price)
);
```

Then type \d at the prompt. You should see the table.

Next, we will enter some data.

```
insert into products (name, price, sale_price) values ('furby', 100, 95);
insert into products (name, price, sale_price)
       values ('frozen lunchbox', 10, 8),
              ('uss enterprise', 12, 11),
              ('spock action figure', 8, 7),
              ('slime', 1, 0.50);
```

Do the following, one at a time.

```
select * from products;
select name, price from products;
select name as product, price as howmuch from products;
```

Discussion...

7 Making Tables

7.1 Creating Tables

We use the CREATE TABLE command. In it's most basic form, it looks like

```
create table NAME (attribute1 type1, attribute2 type2, ...);
```

A simple version of the previous products table is:

```
create table products (
       product_id integer,
       name text,
       price real,
       sale_price real
);
```

This gets the idea, but a few wrinkles are nice. Here's the fancy version again:

```
create table products (
       product_id SERIAL PRIMARY KEY,
       name text,
       price numeric CHECK (price > 0),
       sale_price numeric CHECK (sale_price > 0),
       CHECK (price > sale_price)
);
```

Discussion, including

- Column `product_id` is automatically set when we add a row.
- We have told postgres that `product_id` is the *primary key*.
- Columns `price` and `sale_price` must satisfy some constraints.
- What happens if we try to add data that violates those constraints?

  Try this:

  ```
  insert into products (name, price, sale_price)
         values ('kirk action figure', 50, 52);
  ```

- There are two kinds of constraints here: constraints on *columns* and constraints on the *table*. Which are which?

Here's an alternative approach to making the products table?

```
create table products (
       product_id SERIAL PRIMARY KEY,
       label text UNIQUE NOT NULL CHECK (char_length(label) > 0),
       price numeric CHECK (price >= 0),
       discount numeric DEFAULT 0.0 CHECK (discount >= 0),
       CHECK (price > discount)
);
```

Notice that there are a variety of functions that postgres offers for operating on the different data types. For instance, `char_length()` returns the length of a string.

Now, which one of these will work?

```
insert into products (label, price)
       values ('kirk action figure', 50);
insert into products (price, discount)
       values (50, 42);
insert into products (label, price, discount)
       values ('', 50, 42);
```

7.2 Altering Tables

The `ALTER TABLE` command allows you to change a variety of table features. This includes adding and removing columns, renaming attributes, changing constraints or attribute types, and setting column defaults. See the full documentation for more.

A few examples using the most recent definition of `products`:

- Let's rename `product_id` to just `id` for simplicity.

  ```
  alter table products
        rename product_id to id;
  ```

- Let's add a `brand_name` column.

  ```
  alter table products add brand_name text DEFAULT 'generic' NOT NULL;
  ```

- Let's drop the `discount` column

  ```
  alter table products drop discount;
  ```

- Let's set a default value for `brand_name`.

  ```
  alter table products
        alter brand_name SET DEFAULT 'generic';
  ```

7.3 Deleting Tables

The command is `DROP TABLE`.

```
drop table products;
```

Try it, then type `\d` at the prompt.

8 Working with CRUD

The four most basic operations on our data are

- Create
- Read
- Update
- Delete

collectively known as CRUD operations.

In SQL, these correspond to the four core commands `INSERT`, `SELECT`, `UPDATE`, and `DELETE`.

To start our exploration, let's create a table.

```
create table events (
       id SERIAL PRIMARY KEY,
       moment timestamp DEFAULT 'now',
       persona integer NOT NULL,
       element integer NOT NULL,
       score integer NOT NULL DEFAULT 0 CHECK (score >= 0 and score <= 1000),
       hints integer NOT NULL DEFAULT 0 CHECK (hints >= 0),
       latency real,
       answer text,
       feedback text
);
```

Note: Later on, `persona` and `element` will be foreign keys, but for now, they will just be arbitrary integers.

8.1 INSERT

The basic template is

```
INSERT INTO <tablename> (<column1>, ..., <columnk>)
       VALUES (<value1>, ..., <valuek>)
       RETURNING <expression|*>;
```

where the `RETURNING` clause is optional. If the column names are excluded, then values for all columns must be provided. You can use `DEFAULT` in place of a value for a column with a default setting.

You can also insert multiple rows at once

```
INSERT INTO <tablename> (<column1>, ..., <columnk>)
       VALUES (<value11>, ..., <value1k>),
              (<value21>, ..., <value2k>),
              ...
              (<valuem1>, ..., <valuemk>);
```

8.1.1 Examples

First, copy data from `events.csv` into the events table:

```
\COPY events FROM 'events.csv'
       WITH DELIMITER ',';
SELECT setval('events_id_seq', 1001, false);
```

You should replace the first string by the correct path to the `events.csv` file on your computer.

```
insert into events (persona, element, score, answer, feedback)
       values (1211, 29353, 824, 'C', 'How do the mean and median differ?');
insert into events (persona, element, score, answer, feedback)
       values (1207, 29426, 1000, 'A', 'You got it!')
       RETURNING id;
insert into events (persona, element, score, answer, feedback)
       values (1117, 29433,  842, 'C', 'Try simplifying earlier.'),
              (1199, 29435,    0, 'B', 'Your answer was blank'),
              (1207, 29413, 1000, 'C', 'You got it!'),
              (1207, 29359,  200, 'A', 'A square cannot be negative')
       RETURNING *;
```

Try inserting a few valid rows giving latencies but not id or feedback. Find the value of the id's so inserted.

8.2 SELECT

The `SELECT` command is how we query the database. It is versatile and powerful command.

The simplest query is to look at all rows and columns of a table:

```
select * from events;
```

The `*` is a shorthand for "all columns."

Selects can include expressions, not just column names, as the quantities selected. And we can use `as` clauses to name (or rename) the results.

```
select 1 as one;
select ceiling(10*random()) as r;
select 1 from generate_series(1,10) as ones;
select min(r), avg(r) as mean, max(r) from
       (select random() as r from generate_series(1,10000)) as _;
select timestamp '2015-01-22 08:00:00' + random() * interval '64 days'
       as w from generate_series(1,10);
```

Notice how we used a select to create a virtual table and then selected from it.

Most importantly, we can qualify our queries with conditions that refine the selection. We do this with the `WHERE` clause, which accepts a logical condition on any expression and selects only those rows that satisfy the condition. The conditional expression can include column names (even temporary ones) as variables.

```
select * from events where id > 20 and id < 40;
```

As we will see more next time, we can also order the output using the `ORDER BY` clause and group rows for aggregation using the `GROUP BY` clause values over groups.

```
select score, element from events
    where persona = 1202 order by element, score;
select count(answer) from events where answer = 'A';
select element, count(answer) as numAs
       from events where answer = 'A'
       group by element
       order by numAs;
select persona, avg(score) as mean_score
       from events
       group by persona
       order by mean_score;
```

8.2.1 Examples

Try to craft selects in events for the following:

1. List all event ids for events taking place after 20 March 2015 at 8am. (Hint: `>` and `<` should work as you hope.)

2. List all ids, persona, score where a score $>$ 900 occurred.

3. List all persona (sorted numerically) who score > 900. Can you eliminate duplicates here? (Hint: Consider `SELECT DISTINCT`)
4. Can you guess how to list all persona whose average score > 600. You will need to do a `GROUP BY` as above. (Hint: use `HAVING` instead of `WHERE` for the aggregate condition.)
5. Produce a table showing how many times each instructional element was practiced.

```
select id from events where moment > timestamp '2015-03-20 08:00:00';
select id, persona, score from events where score > 900;
select distinct persona from events where score > 900 order by persona;
select persona from events group by persona having avg(score) > 600;
select element, count(element) from events group by element order by element;
```

8.3 UPDATE

The `UPDATE` command allows us to modify existing entries in any way we like. The basic syntax looks like this

```
UPDATE table
    SET col1 = expression1,
        col2 = expression2,
        ...
    WHERE condition;
```

The `UPDATE` command can update one or more columns and can have a `RETURNING` clause like `INSERT`.

8.3.1 Examples

```
create table gems (label text DEFAULT '',
                   facets integer DEFAULT 0,
                   price money);

insert into gems (select '', ceiling(20*random()+1), money '1.00' from generate_series

update gems set label = ('{thin,quality,wow}'::text[])[ceil(random()*3)];

update gems set label = 'thin'
       where facets < 10;
```

```
update gems set label = 'quality',
                price = 25.00 + cast(10*random() as numeric)
       where facets >= 10 and facets < 20;
update gems set label = 'wow', price = money '100.00'
       where facets >= 20;

select * from gems;
```

Try it:

1. Update events with id $>$ 1000 to set latencies where they are missing. (Consider `select id from events where latency is null;` to find them.)

2. Set answers for id $>$ 1000 to a random letter A through D.

3. Update the scores to subtract 50 points for every hint taken when id $>$ 1000. Check before and after to make sure it worked.

8.4 DELETE

The `DELETE` command allows you to remove rows from a table that satisfy a condition. The basic syntax is:

```
DELETE FROM table WHERE condition;
```

Example:

```
delete from gems where facets < 5;
delete from events where id > 1000 and answer = 'B';
```

Try to delete a few selected rows in one of your existing tables. (Remember: you can do `\d` at the prompt to check the table list.)

9 Activity

Here, we will do some brief practice with CRUD operations by generating a table of random data and playing with it.

1. Create a table `rdata` with five columns: one `integer` column `id`, two `text` columns `a` and `b`, one `date moment`, and one `numeric` column `x`.

2. Use a SELECT command with the generate_series function to display the sequence from 1 to 100.

3. Use a SELECT command with the random() function converted to text (via random()::text) and the md5 function to create a random text string.

4. Use a SELECT command to choose a random element from a fixed array of strings. A fixed text array can be obtained with ('{X,Y,Z}'::text[]) and then indexed using the ceil (ceiling) and random functions to make a selection. (FYI, ('{X,Y,Z}'::text[])[1] would give 'X'.) (SQL is 1-indexed.)

5. SELECT a random date in 2017. You can do this by adding an integer to date '2017-01-01'. For instance, try

   ```
   select date '2017-01-01' + 7 as random_date;
   ```

   For a non-integer type, append ::integer to convert it to an integer.

6. Use INSERT to populate the rdata table with 101 rows, where the id goes from 1 to 100, a is random text, b is random choice from a set of strings (at least three in size), moment contains random days in 2017, and x contains random real numbers in some range.

7. Use SELECT to display rows of the table for which b is equal to a particular choice.

8. Use SELECT with either the ~* or ilike operators to display rows for which a matches a specific pattern, e.g.,

   ```
   select * from rdata where a ~* '[0-9][0-9][a-c]a';
   ```

9. Use SELECT with the overlaps operator on dates to find all rows with moment in the month of November.

10. Use UPDATE to set the value of b to a fixed choice for all rows that are divisible by 3 and 5.

11. Use DELETE to remove all rows for which id is even and greater than 2. (Hint: % is the mod operator.)

12. Use a few more DELETE's (four more should do it) to remove all rows where id is not prime.