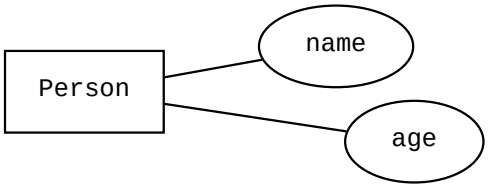
Lab 8

25 Oct 2007

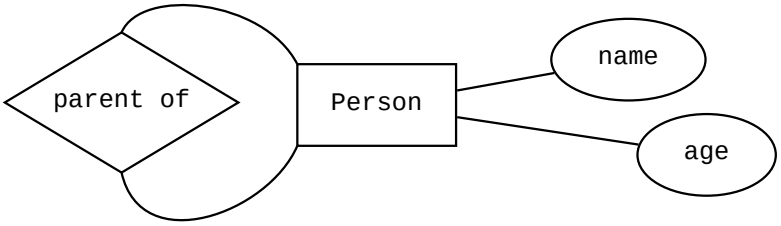
Today we'll use a program called SQLite to set up and query some basic databases using SQL ("Structured Query Language").

The database design

Let's say we want a very simple database that just stores names and ages for a few small families. An entity-relationship diagram (with no relationships, yet) for it might look something like this:



That is, the only entity is a Person, whose attributes are their name and their age. To make things a little more interesting, we can add the relationship "parent of"; that would just involve adding one element to the diagram:



Since Person entities fill both halves of the relationship, we just draw two lines back to the entity set box. (These diagrams often have edges with arrows in them, but for reasons we'll discuss in class later—involving the cardinality of the relationship—this one is correct not to have any.)

Setting up the database

Now, let's play with SQL. To set up an SQL database in SQLite, just pass its name to the program, whose executable is `sqlite3`:

```
sqlite3 family.db
```

This creates the `family.db` file itself, which we can access in subsequent runs just by using the same command line. It is initially empty.

Tables

Although we haven't yet discussed it in lecture, SQL is a language for dealing with *relational databases*. This is a bit different from the format of the E-R diagram (whcich we haven't actually discussed in class in any case), but the database here is small enough to work with in a pretty ad hoc fashion. Eventually we'll talk about more formal methods for converting an E-R diagram into a relational *schema*; for now, the main thing to remember is that everything (everything!) in a relational database is stored in tables, where the columns are different kinds of data (e.g. attributes: name, age, etc) and the rows represent specific instances.

The syntax for creating a table is

> `create table` *name* `(` *columns* `);`

The *name* can be any identifier; the *columns* are a comma-separated list of name-type pairs. The available types are `text`, `integer`, and `real`. Type this at SQL's prompt :

```
create table people (name text, age integer);
```

(Notice that the type comes *after* the identifier here!)

Data

The syntax for entering data in a table is

> `insert into` *table* `values (` *values* `);`

or

> `insert into` *table* `(` *columns* `) values (` *values* `);`

If you just give the values, they're assumed to be in the same order as the columns when the table was created. Or, you can give a list of columns that matches the values you type. Type in these two lines:

```
insert into people values ('Sam', 27);
insert into people (age, name) values (23, 'Alex');
```

Note the single quotes around the strings; also note the order of the pairs in the second line!

Once you've entered some data, you can type

```
.dump
```

(note the initial period) to show a list of commands that would be made to reconstruct all the operations you've done in the current session. They're given in a canonical form—not necessarily the way you originally typed them.

More tables

In the table we've already created, there is no convenient way to also represent the parent relation, so we can add a new table and a few new entries to illustrate it:

```
create table parents (parent text, child text);
insert into people values ('Chris', 50);
insert into parents (parent, child) values ('Chris', 'Alex');
insert into parents (parent, child) values ('Chris', 'Sam');
```

By structuring it this way, we are implicitly making the assumption that the name attribute of each person is a key—that is, a person is uniquely identified by the name we give. In this context (a toy example that will only ever hold two or three families) that's valid, but you should be aware of it.

Task

Enter in a few more people and parent relationships. Hitting up-arrow at the SQL prompt gives you the previous line to edit; just make sure you edit everything that needs to be edited. Make sure you've got at least seven or eight people, of varying age, in at least two distinct families.

Querying

Queries are performed with the `select` keyword. Each query specifies which columns you want from which table:

> `select` *columns* `from` *table* `;`

The *table* is just which table to look in, and *columns* can be either `*` or a comma-separated list of column names:

```
select * from people;
select age,name from people;
```

Most useful queries will limit the results to only those rows of interest, using `where`:

> `select` *columns* `from` *table* `where` *expr* `;`

The *expr* can, among other things, test for equality, inequality, or some boolean combination of other expressions:

```
select * from people where name <> 'Chris';
select name from people where age > 25;
select age from people where age > 20 and name = 'Sam';
```

Note that the "return value" of each `select` statement is a table, even if that table only has one row and one column. This value can be saved by creating a new named table to store it:

> `create table` *name* `as` *select-stmt*

Hence

```
create table ages as select age from people;
```

Remember that you can type `.dump` at the prompt to quickly see everything it's currently storing.

Task

Practice running some queries on the database you encoded earlier. Can you get just a list of Chris's children? All the teenagers?

You might want to do something like list the ages of Chris's children. Think for a moment about what that would entail, and then move on to the next section.

More, larger queries

In a browser window, load up the page `http://www.sqlite.org/lang.html` (also linked from the course homepage). This page gives the entire subset of SQL that is understood by SQLite3, both the syntax and the semantics. So far you've seen two commands: `create table` and `select`. Click on the link to SELECT and you'll see all the different syntactic possibilities for a `select` statement; below the grammar you'll see the semantics underlying all the variants.

Type the following line into SQLite:

```
select * from (select parent, child as name from parents) natural join people;
```

Use the documentation page to figure out what the heck is happening in this query (and how to interpret its output).

Task

Now can you write a query that retrieves a list of the ages of Chris's children? How about a table that associates all *grand*parents with their grandkids (after entering data that would make this query relevant, of course)?

(UI note: remember that "readline" library Chris mentioned on Monday? You're using it right now for the SQLite command prompt. Not only can you up-arrow to scroll through previous typed-in lines, you can also use the Home and End keys (or their old-school equivalents `^A` and `^E`) to go to the beginning and end of the line, as well as a bunch of other convenient, standard editing shortcuts.)