Lab 9

1 Nov 2007

More fun with SQL! Today you'll see how to convert the mathematical notation of the relational algebra into specific SQL queries.

First, some data

Fire up a fresh SQLite database and input the following tables:

Player:

name	token	bank
Sam	shoe	$342
Alex	wheelbarrow	$1,230
Chris	car	$37

Owes:

creditor	debtor	amount
Sam	Chris	$350
Alex	Chris	$100

Simple queries

In SQL, pretty much all the operators are bundled up into the single `select` command, which we saw briefly last week.

Operation	**To do this:**	**Type this:**
Union	$R \cup S$	`select * from R union select * from S`
Intersection	$R \cap S$	`select * from R intersect select * from S`
Complement	$R - S$	`select * from R except select * from S`
Product	$R \times S$	`select * from R, S`
Projection	$\pi_{C,D}(R)$	`select C,D from R`
Renaming	$\pi_{C \to D}(R)$	`select C as D from R`
Combination	$\pi_{C+D \to E}(R)$	`select C + D as E from R`
Selection	$\sigma_{C<10}(R)$	`select * from R where C < 10`
Join	$R \bowtie S$	`select * from R natural join S`
Group	$\gamma_{C,\Sigma(D)}(R)$	`select C, SUM(D) from R group by C`
Sort	$\tau_{C,D}(R)$	`select * from R order by C,D`

(Join was just explained briefly at the end of class; I expect you to experi-

ment with it to see how it works. Remember, its use requires that the relevant tables have (some) matching column labels! Group and sort I didn't talk about, but this is actually a case where SQL's English-like syntax can be helpful: they pretty much do what they say they do.)

Try writing the following queries:

- The names of all players
- The data for all players with more than $50 in the bank
- The total debt load of each debtor
- The data for all players, sorted by amount in the bank

Complex queries

Generally, if two different queries are accomplished in different parts of a `select` statement, they can be combined into a single statement. For instance,

$$\pi_{C,D}(\sigma_{E \geq 0}(R))$$

is written simply as

```
select C, D from R where E >= 0;
```

Where this is not possible, two strategies can be employed. One is naming intermediate values:

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

or

`create view` *name* `as` *select-stmt*

Then *name* can be used in subsequent queries. The chief difference is whether the new data is copied or not; in the CREATE TABLE form, the query is run and an actual new table is stored. In the CREATE VIEW form, the query is stored so that in the future you can refer to the view as if it were a table, but subsequent changes to the tables used in the *select-stmt* will also be reflected in the named view.

Alternatively, most places you can use a table name R, you can also equivalently use a whole other select statement (which you may need to put in parentheses):

`select * from R natural join (` *select-stmt* `);`

Tips: Use `.dump` frequently to see what is in each table. (Also useful: `.mode` and `.schema`. Type `.help` for more info.) If you build up a very complicated expression and SQLite starts complaining about ambiguous labels, you can further specify the column header by which table it came from—`Player.name` instead of just `name`, for instance. The expressions that can be used in Combination and in Grouping are fairly arbitrary—see `http://www.sqlite.org/lang_expr.html` for a complete list.

Try to build a query to return a table `NetWorth` that records the total assets of each player: the amount they have in front of them, plus their credits, minus their debits.

Hang on to this little toy database and feel free to augment it; small examples like this one are really the best way to figure out the nuances of the different kinds of SELECT statements. Once you've got `NetWorth` working, though, you can go ahead and work on the project.

A project note: remember that the SQL driver we're using is SQLite-based; the actual file it creates is in the SQLite format. That means that after a run of your program, you can use the SQLite command line executable to inspect and interact with the database that got written out, which could be very useful in debugging!