61A Lecture 33

Wednesday, November 19

Announcements

• Project 4 due Friday 11/21 @ 11:59pm
• Early submission point #3: Submit by Thursday 11/20 @ 11:59pm
• Homework 9 (6 pts) due Wednesday 11/26 @ 11:59pm
• Guest in live lecture, TA Soumya Basu, on Monday 11/24 (videos still by John)
• No lecture on Wednesday 11/26 (turkey)

Numerical Expressions

Expressions can contain function calls and arithmetic operators

\[
\text{select} \ [\text{columns}] \ \text{from} \ \text{table} \ \text{where} \ [\text{expression}] \ \text{order by} \ [\text{expression}];
\]

Combine values: +, -, *, /, %, and, or
Transform values: abs, round, not, -
Compare values: <, <=, >, >=, <>, !=, =

(string)

String Expressions

String values can be combined to form longer strings

```sql
sqlite> select "hello, world" as s;
hello, world
```

Basic string manipulation is built into SQL, but differs from Python

```sql
sqlite> create table phrase as select "hello, world" as s;
sqlite> select substr(s, 4, 2) || substr(s, instr(s, " ")+1, 1) from phrase;
```

Strings can be used to represent structured values, but doing so is rarely a good idea

```sql
sqlite> create table lists as select "one" as car, "two,three,four" as cdr;
```

(sql)

Useful Python Features

The namedtuple function returns a new sub-class of tuple

```python
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
            City(42, 71, "Cambridge"),
            City(44, 91, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

Attribute names are accessible as the _fields attribute of an instance

```python
>>> print(cities[0])
City(latitude=38, longitude=122, name='Berkeley')
```

```python
>>> print(cities[0]._fields)
('latitude', 'longitude', 'name')
```

The eval function can take a dictionary of name-value bindings as a second argument

```python
>>> eval("(latitude + 3)",
       {"latitude": 38})
41
```

(sql)
A Select Class

The SQL parser creates an instance of the Select class for each select statement

```python
>>> class Select:
    """select [columns] from [tables] where [condition]."""
    def __init__(self, columns, tables, condition):
        self.columns = columns
        self.tables = tables
        self.condition = condition
        self.make_row = create_make_row(self.columns)
    def execute(self, env):
        """Join, filter, and map rows from tables to columns."""
        from_rows = product(self.tables, env)
        filtered_rows = filter(self.filter_fn, from_rows)
        return map(self.make_row, filtered_rows)
    def filter_fn(self, row):
        if self.condition:
            return eval(self.condition, env)
        else:
            return True
```

Simplified version of http://composingprograms.com/examples/sql/sql_exec.py

Interpreting Select Statements

Creating Row Classes Dynamically

Each select statement creates a table with new columns, represented by a new class

```python
>>> def create_make_row(description):
    """Return a function from an input environment (dict) to an output row."
    description -- a comma-separated list of [expression] as [column name]
    columns -- a column.split("")
    expressions, names = [], []
    for column in columns:
        if " as " in column:
            expression, name = column.split(" as ")
            else:
                expression, name = column, column
                expressions, names.append(name)
        return namedtuple(name, names)
    def make_env(rows, names):
        """Create an environment of names bound to values."""
        env = dict(zip(names, rows))
        for name in row._fields:
            env[name] = getattr(row, name)
        return env
```

SQL Interpreter Examples

A Select Statement Filters, Sorts, and Maps Rows

One correct (but not always efficient) implementation of select uses sequence operations

```python
>>> select name, 60*abs(latitude-38) as distance from cities where name in "Berkeley";
>>> select 42, 72, "Cambridge" union select 93, "Minneapolis"
>>> select 45(lat-38) as north from cities where name in "Berkeley"
```

Joining Rows

Joining creates a dictionary with all names and aliases for each combination of rows

```python
>>> def join(tables, env):
    """Return an iterator over dictionaries from names to values in a row."
    names = tables.split("")
    joined_rows = product(ei(env[name] for name in names)
    return map(lambda row: make_env(rows, names), joined_rows)
>>> def make_env(rows, names):
    """Create an environment of names bound to values."""
    env = dict(zip(names, rows))
    for name in row._fields:
        env[name] = getattr(row, name)
    return env
```

Interpreting SQL Using Python

Fill in the blanks in this interactive Python session that interprets these SQL statements

```python
create table cities as
select 38 as lat, 122 as lon, "Berkeley" as name union
select 42, 72, "Cambridge" union
select 93 "Minneapolis"
select 60*lat-38 as north from cities where name in "Berkeley"
```

Database Management Systems

```python
>>> select name, 60*abs(latitude-38) as distance from cities where name in "Berkeley"
```

Expression from column description

```
Name from column description
```
The manner in which tables are filtered, sorted, and joined affects execution time.

Select the parents of curly-furred dogs:

```sql
select parent from parents, dogs
where child = name and fur = "curly"
```

Join all rows of parents to all rows of dogs, filter by `child = name` and `fur = "curly"`

Join only rows of parents and dogs where `child = name`, filter by `fur = "curly"`

Filter dogs by `fur = "curly"`, join result with all rows of parents, filter by `child = name`

Filter dogs by `fur = "curly"`, join only rows of result and parents where `child = name`