# 61A Lecture 33

Wednesday, November 19

## Announcements

## Announcements

- `Project 4 due Friday 11/21 @ 11:59pm`

## Announcements

- Project 4 due Friday 11/21 @ 11:59pm
  - Early submission point #3: Submit by Thursday 11/20 @ 11:59pm

## 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

## 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)

## 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

## Numerical Expressions

Expressions can contain function calls and arithmetic operators

## Numerical Expressions

Expressions can contain function calls and arithmetic operators

```
select [columns] from [table] where [expression] order by [expression];
```

## Numerical Expressions

Expressions can contain function calls and arithmetic operators

```
[expression] as [name], [expression] as [name], ...

select [columns] from [table] where [expression] order by [expression];
```

## Numerical Expressions

Expressions can contain function calls and arithmetic operators

```
                  [expression] as [name], [expression] as [name], ...

            select [columns] from [table] where [expression] order by [expression];
```

Combine values: +, -, *, /, %, and, or

## Numerical Expressions

Expressions can contain function calls and arithmetic operators

```
        [expression] as [name], [expression] as [name], ...

select [columns] from [table] where [expression] order by [expression];
```

Combine values: +, -, *, /, %, and, or

Transform values: abs, round, not, -

## Numerical Expressions

Expressions can contain function calls and arithmetic operators

```
                [expression] as [name], [expression] as [name], ...

          select [columns] from [table] where [expression] order by [expression];
```

Combine values: +, -, *, /, %, and, or

Transform values: abs, round, not, -

Compare values: <, <=, >, >=, <>, !=, =

## Numerical Expressions

Expressions can contain function calls and arithmetic operators

```
[expression] as [name], [expression] as [name], ...

select [columns] from [table] where [expression] order by [expression];
```

Combine values: +, -, *, /, %, and, or

Transform values: abs, round, not, -

Compare values: <, <=, >, >=, <>, !=, =

(Demo)

## String Expressions

## String Expressions

String values can be combined to form longer strings

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

```
sqlite> create table phrase as select "hello, world" as s;
```

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

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

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

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

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

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

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

```
sqlite> create table lists as select "one" as car, "two,three,four" as cdr;
sqlite> select substr(cdr, 1, instr(cdr, ",")-1) as cadr from lists;
```

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

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

```
sqlite> create table lists as select "one" as car, "two,three,four" as cdr;
sqlite> select substr(cdr, 1, instr(cdr, ",")-1) as cadr from lists;
two
```

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

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

```
sqlite> create table lists as select "one" as car, "two,three,four" as cdr;
sqlite> select substr(cdr, 1, instr(cdr, ",")-1) as cadr from lists;
two
```

## String Expressions

String values can be combined to form longer strings

```
sqlite> select "hello," || " world";
hello, world
```

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

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

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

```
sqlite> create table lists as select "one" as car, "two,three,four" as cdr;
sqlite> select substr(cdr, 1, instr(cdr, ",")-1) as cadr from lists;
two
```

(Demo)

## SQL Execution

## Useful Python Features

## Useful Python Features

```
The namedtuple function
returns a new sub-class
of tuple
```

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
```

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
```

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

Attribute names are accessible as the _fields attribute of an instance of City

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

Attribute names are accessible as the _fields attribute of an instance of City

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

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

Attribute names are accessible as the _fields attribute of an instance of City

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

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

Attribute names are accessible as the _fields attribute of an instance of City

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

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

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

Attribute names are accessible as the _fields attribute of an instance of City

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

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

```
>>> eval("latitude + 3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'latitude' is not defined
```

## Useful Python Features

The namedtuple function returns a new sub-class of tuple

```
>>> from collections import namedtuple
>>> City = namedtuple("City", ["latitude", "longitude", "name"])
>>> cities = [City(38, 122, "Berkeley"),
              City(42,  71, "Cambridge"),
              City(43,  93, "Minneapolis")]
>>> [city.latitude for city in cities]
[38, 42, 43]
```

Attribute names are accessible as the _fields attribute of an instance of City

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

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

```
>>> eval("latitude + 3")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError: name 'latitude' is not defined
>>> eval("latitude + 3", {"latitude": 38})
41
```

## A Select Statement Filters, Sorts, and Maps Rows

## A Select Statement Filters, Sorts, and Maps Rows

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

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
```

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
>>> def columns(city):
        latitude, longitude, name = city
        return Distance(name, 60*abs(latitude-38))
```

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
>>> def columns(city):
        latitude, longitude, name = city
        return Distance(name, 60*abs(latitude-38))
>>> def condition(city):
        latitude, longitude, name = city
        return name != "Berkeley"
```

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
>>> def columns(city):
        latitude, longitude, name = city
        return Distance(name, 60*abs(latitude-38))
>>> def condition(city):
        latitude, longitude, name = city
        return name != "Berkeley"
>>> for row in map(columns, filter(condition, cities)):
        print(row)
```

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
>>> def columns(city):
        latitude, longitude, name = city
        return Distance(name, 60*abs(latitude-38))
>>> def condition(city):
        latitude, longitude, name = city
        return name != "Berkeley"
>>> for row in map(columns, filter(condition, cities)):
        print(row)
Row(name='Miami', distance=720)
...
```

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
>>> def columns(city):
        latitude, longitude, name = city
        return Distance(name, 60*abs(latitude-38))
>>> def condition(city):
        latitude, longitude, name = city
        return name != "Berkeley"
>>> for row in map(columns, filter(condition, cities)):
        print(row)
Row(name='Miami', distance=720)
...
```

Names from column description

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
>>> def columns(city):
        latitude, longitude, name = city
        return Distance(name, 60*abs(latitude-38))
>>> def condition(city):
        latitude, longitude, name = city
        return name != "Berkeley"
>>> for row in map(columns, filter(condition, cities)):
        print(row)
Row(name='Miami', distance=720)
...
```

Names from column description

Expressions from column description

## A Select Statement Filters, Sorts, and Maps Rows

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

```
sqlite> select name, 60*abs(latitude-38) as distance from cities where name != "Berkeley";
Miami|720
San Diego|300
Cambridge|240
Minneapolis|420
North Pole|3120
```

```
>>> Distance = namedtuple("Row", ["name", "distance"])
>>> def columns(city):
        latitude, longitude, name = city
        return Distance(name, 60*abs(latitude-38))
>>> def condition(city):
        latitude, longitude, name = city
        return name != "Berkeley"
>>> for row in map(columns, filter(condition, cities)):
        print(row)
Row(name='Miami', distance=720)
...
```

Names from column description

Expressions from column description

## Interpreting Select Statements

## A Select Class

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

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

## A Select Class

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

```
>>> class Select:
        """select [columns] from [tables] where [condition]."""
```

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

## A Select Class

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

```
>>> 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)
```

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

## A Select Class

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

```
>>> 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 = join(self.tables, env)
            filtered_rows = filter(self.filter_fn, from_rows)
            return map(self.make_row, filtered_rows)
```

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

## A Select Class

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

```
>>> 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 = join(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, row)
            else:
                return True
```

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

## Creating Row Classes Dynamically

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

## Creating Row Classes Dynamically

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

```
>>> 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]
        """
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
        expressions, names = [], []
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
        expressions, names = [], []
        for column in columns:
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
        expressions, names = [], []
        for column in columns:
            if " as " in column:
                expression, name = column.split(" as ")
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
        expressions, names = [], []
        for column in columns:
            if " as " in column:
                expression, name = column.split(" as ")
            else:
                expression, name = column, column
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
        expressions, names = [], []
        for column in columns:
            if " as " in column:
                expression, name = column.split(" as ")
            else:
                expression, name = column, column
            expressions.append(expression)
            names.append(name)
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
        expressions, names = [], []
        for column in columns:
            if " as " in column:
                expression, name = column.split(" as ")
            else:
                expression, name = column, column
            expressions.append(expression)
            names.append(name)
        row = namedtuple("Row", names)
```

## Creating Row Classes Dynamically

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

```
>>> 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 = description.split(", ")
        expressions, names = [], []
        for column in columns:
            if " as " in column:
                expression, name = column.split(" as ")
            else:
                expression, name = column, column
            expressions.append(expression)
            names.append(name)
        row = namedtuple("Row", names)
        return lambda env: row(*[eval(e, env) for e in expressions])
```

## Joining Rows

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

(Demo)

## Joining Rows

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

```
>>> from itertools import product
```

(Demo)

## Joining Rows

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

```
>>> from itertools import product
>>> def join(tables, env):
        """Return an iterator over dictionaries from names to values in a row."""
        names = tables.split(", ")
        joined_rows = product(*[env[name] for name in names])
        return map(lambda rows: make_env(rows, names), joined_rows)
```

(Demo)

## Joining Rows

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

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

(Demo)

## Joining Rows

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

```
>>> from itertools import product
>>> def join(tables, env):
        """Return an iterator over dictionaries from names to values in a row."""
        names = tables.split(", ")
        joined_rows = product(*[env[name] for name in names])
        return map(lambda rows: 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 row in rows:
            for name in row._fields:
                env[name] = getattr(row, name)
```

(Demo)

## Joining Rows

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

```
>>> from itertools import product
>>> def join(tables, env):
        """Return an iterator over dictionaries from names to values in a row."""
        names = tables.split(", ")
        joined_rows = product(*[env[name] for name in names])
        return map(lambda rows: 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 row in rows:
            for name in row._fields:
                env[name] = getattr(row, name)
        return env
```

(Demo)

## SQL Interpreter Examples

## Interpreting SQL Using Python

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

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,        71,         "Cambridge"        union
  select 45,        93,         "Minneapolis";
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,        71,         "Cambridge"        union
  select 45,        93,         "Minneapolis";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,        71,         "Cambridge"        union
  select 45,        93,         "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,         71,        "Cambridge"       union
  select 45,         93,        "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select(________________________, ________________, _____________________________________)

>>> for row in s.execute(____________________________________________________________________):
...     print(row)
...
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,        71,         "Cambridge"        union
  select 45,        93,         "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select(________________________, ________________, _____________________________________)

>>> for row in s.execute(________________________________________________________________________):
...     print(row)
...
Row(north=240)
Row(north=300)
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,        71,         "Cambridge"        union
  select 45,        93,         "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select(________________________, ________________, _____________________________________)

>>> for row in s.execute(____________________________________________________________________):
...     print(row)
...
Row(north=240)
Row(north=300)
```

```
>>> class Select:
        """select [columns] from [tables] where [condition]."""
        def __init__(self, columns, tables, condition):
            ...
        def execute(self, env):
            ...
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,         71,        "Cambridge"        union
  select 45,         93,        "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select('60*(lat-38) as north', ________________, _____________________________________)

>>> for row in s.execute(________________________________________________________________________):
...     print(row)
...
Row(north=240)
Row(north=300)
```

```
>>> class Select:
        """select [columns] from [tables] where [condition]."""
        def __init__(self, columns, tables, condition):
            ...
        def execute(self, env):
            ...
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,         71,        "Cambridge"        union
  select 45,         93,        "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select('60*(lat-38) as north', 'cities', ______________________)

>>> for row in s.execute(____________________________________):
...     print(row)
...
Row(north=240)
Row(north=300)
```

```
>>> class Select:
        """select [columns] from [tables] where [condition]."""
        def __init__(self, columns, tables, condition):
            ...
        def execute(self, env):
            ...
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,        71,         "Cambridge"        union
  select 45,        93,         "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select('60*(lat-38) as north', 'cities', 'name != "Berkeley"')

>>> for row in s.execute(______________________________):
...     print(row)
...
Row(north=240)
Row(north=300)
```

```
>>> class Select:
        """select [columns] from [tables] where [condition]."""
        def __init__(self, columns, tables, condition):
            ...
        def execute(self, env):
            ...
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,         71,        "Cambridge"        union
  select 45,         93,        "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select('60*(lat-38) as north', 'cities', 'name != "Berkeley"')

>>> for row in s.execute({"cities": cities}):
...     print(row)
...
Row(north=240)
Row(north=300)
```

```
>>> class Select:
        """select [columns] from [tables] where [condition]."""
        def __init__(self, columns, tables, condition):
            ...
        def execute(self, env):
            ...
```

## Interpreting SQL Using Python

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

```
create table cities as
  select 38 as lat, 122 as lon, "Berkeley" as name union
  select 42,         71,        "Cambridge"       union
  select 45,         93,        "Minneapolis";
select 60*(lat-38) as north from cities where name != "Berkeley";
```

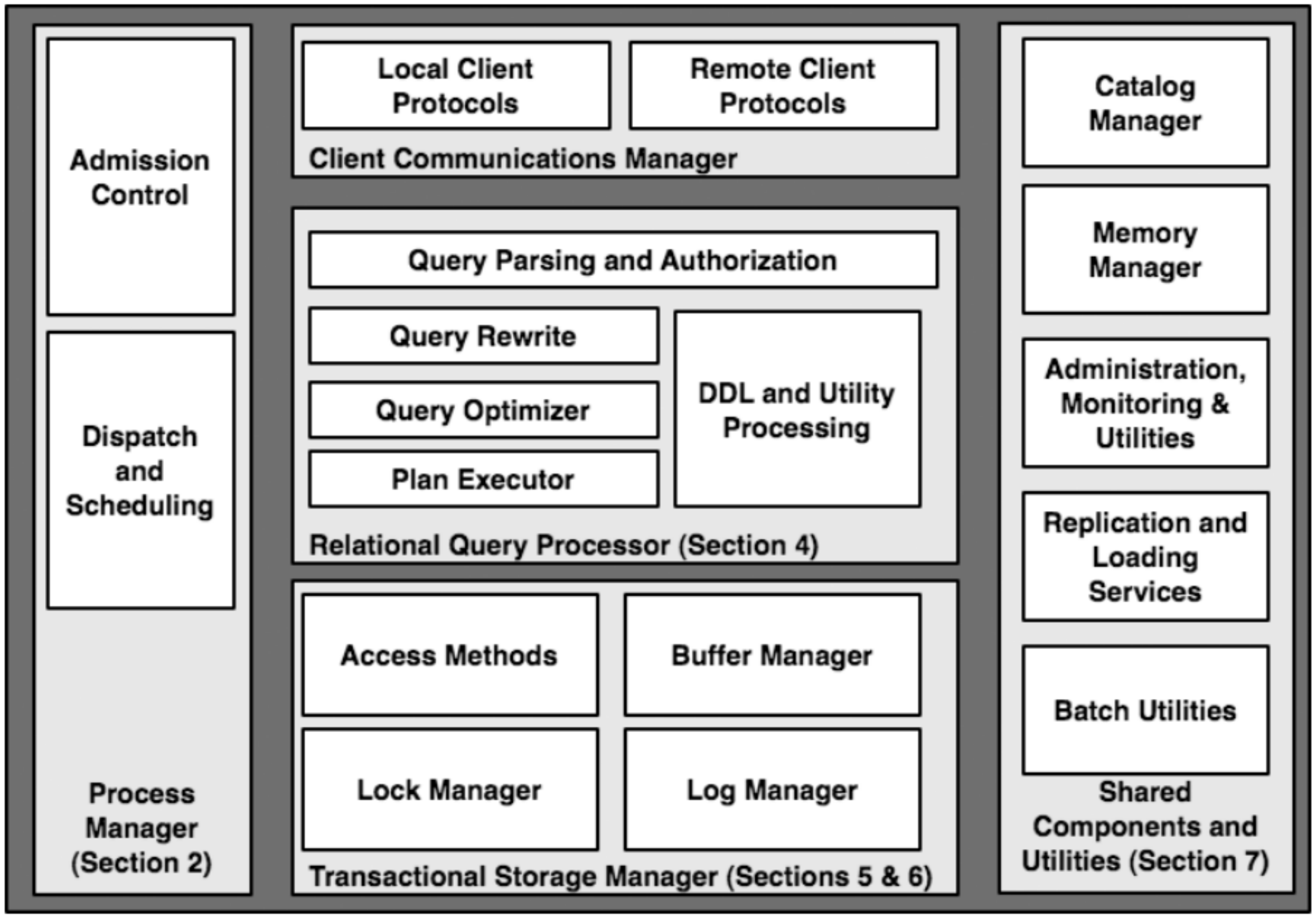
```
>>> City = namedtuple("City", ["lat", "lon", "name"])
>>> cities = [City(38, 122, "Berkeley"), City(42,  71, "Cambridge"), City(43,  93, "Minneapolis")]

>>> s = Select('60*(lat-38) as north', 'cities', 'name != "Berkeley"')

>>> for row in s.execute({"cities": cities}):
...     print(row)
...
Row(north=240)
Row(north=300)
```

How many times is eval called during this call to s.execute? (Demo)

```
>>> class Select:
        """select [columns] from [tables] where [condition]."""
        def __init__(self, columns, tables, condition):
            ...
        def execute(self, env):
            ...
```

## Database Management Systems

## Database Management System Architecture



Admission Control
Dispatch and Scheduling
Process Manager (Section 2)
Local Client Protocols
Remote Client Protocols
Client Communications Manager
Query Parsing and Authorization
Query Rewrite
Query Optimizer
Plan Executor
DDL and Utility Processing
Relational Query Processor (Section 4)
Access Methods
Buffer Manager
Lock Manager
Log Manager
Transactional Storage Manager (Sections 5 & 6)
Catalog Manager
Memory Manager
Administration, Monitoring & Utilities
Replication and Loading Services
Batch Utilities
Shared Components and Utilities (Section 7)

`Architecture of a Database System by Hellerstein, Stonebreaker, and Hamilton`

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

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

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

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

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

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

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

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

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

```
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"

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

```
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"

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

```
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

## Query Planning

The manner in which tables are filtered, sorted, and joined affects execution time

Select the parents of curly-furred dogs:

```
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