Advanced Data Management (CSCI 640/490)

Databases

Dr. David Koop

Exercise

- Given variables `x` and `y`, print the long division answer of `x` divided by `y` with the remainder.
- Examples:
  - `x = 11, y = 4` should print `"2R3"`
  - `x = 15, y = 2` should print `"7R1"`

Exercise

- Suppose I want to write Python code to print the numbers from 1 to 100. What errors do you see?

```
// print the numbers from 1 to 100
int counter = 1
while counter < 100 {
    print counter
    counter++
}
```

Exercise

- Suppose `a = ['a', 'b', 'c', 'd']` and `b = (1, 2, 3)`
- What happens with?
  - `a[0]`
  - `a[1:2]`
  - `b[:-2]`
  - `b.append(4)`
  - `a.extend(b)`
  - `a.pop(0)`
  - `b[0] = "100"`
  - `b + (4,)`

Exercise

- Suppose `a = ['a', 'b', 'c', 'd']` and `b = (1, 2, 3)`
- What happens with?
  - `a[0] # 'a'`
  - `a[1:2] # ['b']`
  - `b[:-2] # (1,)`
  - `b.append(4) # error`
  - `a.extend(b) # ['a', 'b', 'c', 'd', 1, 2, 3]`
  - `a.pop(0) # 'a' with side effect a becomes ['b', 'c', 'd']`
  - `b[0] = "100" # error`
  - `b + (4,) # (1,2,3,4)`

Example: Counting Letters

- Write code that takes a string `s` and creates a dictionary with that counts how often each letter appears in `s`
- `count_letters("Mississippi") →`
  `{'s': 4, 'i': 4, 'p': 2', 'M': 1}`

Python Containers

- Container: store more than one value
- Mutable versus immutable: Can we update the container?
  - Yes → mutable
  - No → immutable
  - Lists are mutable, tuples are immutable
- Lists and tuples may contain values of different types:
- List: `[1,"abc",12.34]`
- Tuple: `(1, "abc", 12.34)`
- You can also put functions in containers!
- `len` function: number of items: `len(l)`

Indexing and Slicing

- Strings and collections are the same
- Indexing:
  - Where do we start counting?
  - Use brackets `[]` to retrieve one value
  - Can use negative values (count from the end)
- Slicing:
  - Use brackets plus a colon to retrieve multiple values:

    `[<start>:<end>]`
  - Returns a **new** list (`b = a[:]`)
  - Don't need to specify the beginning or end

Dictionaries

- One of the most useful features of Python
- Also known as associative arrays
- Exist in other languages but a core feature in Python
- Associate a key with a value
- When I want to find a value, I give the dictionary a key, and it returns the value
- Example: InspectionID (key) → InspectionRecord (value)
- Keys must be immutable (technically, hashable):
  - Normal types like numbers, strings are fine
  - Tuples work, but lists do not (TypeError: unhashable type: 'list')
- There is only one value per key!

Sets

- Sets are like dictionaries but without any values:
- `s = {'MA', 'RI', 'CT', 'NH'}; t = {'MA', 'NY', 'NH'}`
- `{}` is an empty dictionary, `set()` is an empty set
- Adding values: `s.add('ME')`
- Removing values: `s.discard('CT')`
- Exists: `"CT" in s`
- Union: `s | t => {'MA', 'RI', 'CT', 'NH', 'NY'}`
- Intersection: `s & t => {'MA', 'NH'}`
- Exclusive-or (xor): `s ^ t => {'RI', 'CT', 'NY'}`
- Difference: `s - t => {'RI', 'CT'}`

Assignment 1

- Due Monday
- Using Python for data analysis on MoMA data
- Use basic python for now to work on language knowledge
- Use Anaconda or a hosted Python environment
- Turn `.ipynb` file in via Blackboard

Nesting Containers

- Can have lists inside of lists, tuples inside of tuples, dictionaries inside of dictionaries
- Can also have dictionaries inside of lists, tuples inside of dictionaries, …
- ```
  d = {"Brady": [(2015, 4770, 36), (2014, 4109, 33)],
       "Luck": [(2015, 1881, 15), (2014, 4761, 40)],
       …
      }
  ```
- JavaScript Object Notation (JSON) looks very similar for literal values; Python allows variables in these types of structures
  ```

Nesting Code

- Can have loops inside of loops, if statements inside of if statements
- Careful with variable names:
-
```
l = {0: 0, 1: 3, 4: 5, 9: 12}
for i in range(100):
    square = i ** 2
    max_val = l[square]
    for i in range(max_val):
        print(i)
```
- Strange behavior, likely unintended, but Python won't complain!

None

- Like null in other languages, used as a placeholder when no value exists
- The value returned from a function that doesn't return a value

```
def f(name):
    print("Hello,", name)
v = f("Patricia") # v will have the value None
```

- Also used when you need to create a new list or dictionary:

```
def add_letters(s, d=None):
    if d is None:
        d = {}
    d.update(count_letters(s))
```

- Looks like `d={}` would make more sense, but that causes issues
- `None` serves as a **sentinel** value in `add_letters`

is and ==

- `==` does a normal equality comparison
- `is` checks to see if the object is the exact same object
- Common style to write statements like `if d is None: …`
- Weird behavior:
  - ```
    a = 4 - 3
    a is 1 # True
    ```
  - ```
    a = 10 ** 3
    a is 1000 # False
    ```
  - ```
    a = 10 ** 3
    a == 1000 # True
    ```
- Generally, avoid `is` unless writing `is None`
    ```

is and ==

- `==` does a normal equality comparison
- `is` checks to see if the object is the exact same object
- Common style to write statements like `if d is None: …`
- Weird behavior:
  - ```
    a = 4 - 3
    a is 1 # True
    ```
  - ```
    a = 10 ** 3
    a is 1000 # False
    ```
  - ```
    a = 10 ** 3
    a == 1000 # True
    ```

Python caches common integer objects

- Generally, avoid `is` unless writing `is None`
    ```

Objects

- `d = dict() # construct an empty dictionary object`
- `l = list() # construct an empty list object`
- `s = set() # construct an empty set object`
- `s = set([1,2,3,4]) # construct a set with 4 numbers`
- Calling methods:
  - `l.append('abc')`
  - `d.update({'a': 'b'})`
  - `s.add(3)`
- The method is tied to the object preceding the dot (e.g. `append` modifies `l` to add `'abc'`)

Python Modules

- Python module: a file containing definitions and statements
- Import statement: like Java, get a module that isn't a Python builtin

```
import collections
d = collections.defaultdict(list)
d[3].append(1)
```

- `import <name> as <shorter-name>`

```
import collections as c
```

- `from <module> import <name>` : don't need to refer to the module

```
from collections import defaultdict
d = defaultdict(list)
d[3].append(1)
```

Other Collections

- `collections.defaultdict`: specify a default value for any item in the dictionary (instead of `KeyError`)
- `collections.OrderedDict`: keep entries ordered according to when the key was inserted
    - `dict` objects are ordered in Python 3.7 but `OrderedDict` has some other features (equality comparison, reversed)
- `collections.Counter`: counts hashable objects, has a `most_common` method

Example: Counting Letters

- Write code that takes a string `s` and creates a dictionary with that counts how often each letter appears in `s`
- `count_letters("Mississippi") →`
  `{'s': 4, 'i': 4, 'p': 2', …}`

Solution using Counter

- Use an existing library made to count occurrences

```
from collections import Counter
Counter("Mississippi")
```

- produces

```
Counter({'M': 1, 'i': 4, 's': 4, 'p': 2})
```

- Improve: convert to lowercase first

Iterators

- Remember `range, values, keys, items`?
- They return **iterators**: objects that traverse containers
- Given iterator `it`, `next(it)` gives the next element
- `StopIteration` exception if there isn't another element
- Generally, we don't worry about this as the for loop handles everything automatically…but you cannot index or slice an iterator
- `d.values()[0]` will not work!
- If you need to index or slice, construct a list from an iterator
- `list(d.values())[0]` or `list(range(100))[-1]`
- In general, this is slower code so we try to avoid creating lists

List Comprehensions

- Shorthand for transformative or filtering for loops
- ```
  squares = []
  for i in range(10):
      squares.append(i**2)
  ```
- `squares = [i**2 for i in range(10)]`
- Filtering:
- ```
  squares = []
  for i in range(10):
      if i % 3 != 1:
          squares.append(i ** 2)
  ```
- `squares = [i**2 for i in range(10) if i % 3 != 1]`
- if clause **follows** the for clause

Dictionary Comprehensions

- Similar idea, but allow dictionary construction
- Could use lists:
  - `names = dict([(k, v) for k,v in … if …])`
- Native comprehension:
  - ```
    names = {"Al": ["Smith", "Brown"], "Beth":["Jones"]}
    first_counts ={k: len(v) for k,v in names.items()}
    ```
- Could do this with a for loop as well
    ```

Exceptions

- errors but potentially something that can be addressed
- try-except-else-finally:
  - `except` clause runs if exactly the error(s) you wish to address happen
  - `else` clause will run if no exceptions are encountered
  - `finally` always runs (even if the program is about to crash)
- Can have multiple `except` clauses
- can also raise exceptions using the `raise` keyword
- (and define your own)

Classes

- `class ClassName:`
  `    …`
- Everything in the class should be indented until the declaration ends
- `self`: `this` in Java or C++ is `self` in Python
- Every instance method has `self` as its first parameter
- Instance variables are defined **in methods** (usually constructor)
- `__init__`: the constructor, should initialize instance variables
- ```
  def __init__(self):
      self.a = 12
      self.b = 'abc'
  ```
- ```
  def __init__(self, a, b):
      self.a = a
      self.b = b
  ```

Class Example

- ```
  class Rectangle:
      def __init__(self, x, y, w, h):
          self.x = x
          self.y = y
          self.w = w
          self.h = h

      def set_corner(self, x, y):
          self.x = x
          self.y = y

      def set_width(self, w): self.w = w

      def set_height(self, h): self.h = h

      def area(self):
          return self.w * self.h
  ```
  ```

Databases

Database

- Basically, just structured data/information stored on a computer
- Very generic, doesn't specify specific way that data is stored
- Can be single-file (or in-memory) or much more complex
- Methods to:
  - add, update, and remove data
  - query the data

Using Databases

- Suppose we just use a single file or a set of files to store data
- Now, we write programs to use that data
- What are the potential issues?

Using Databases

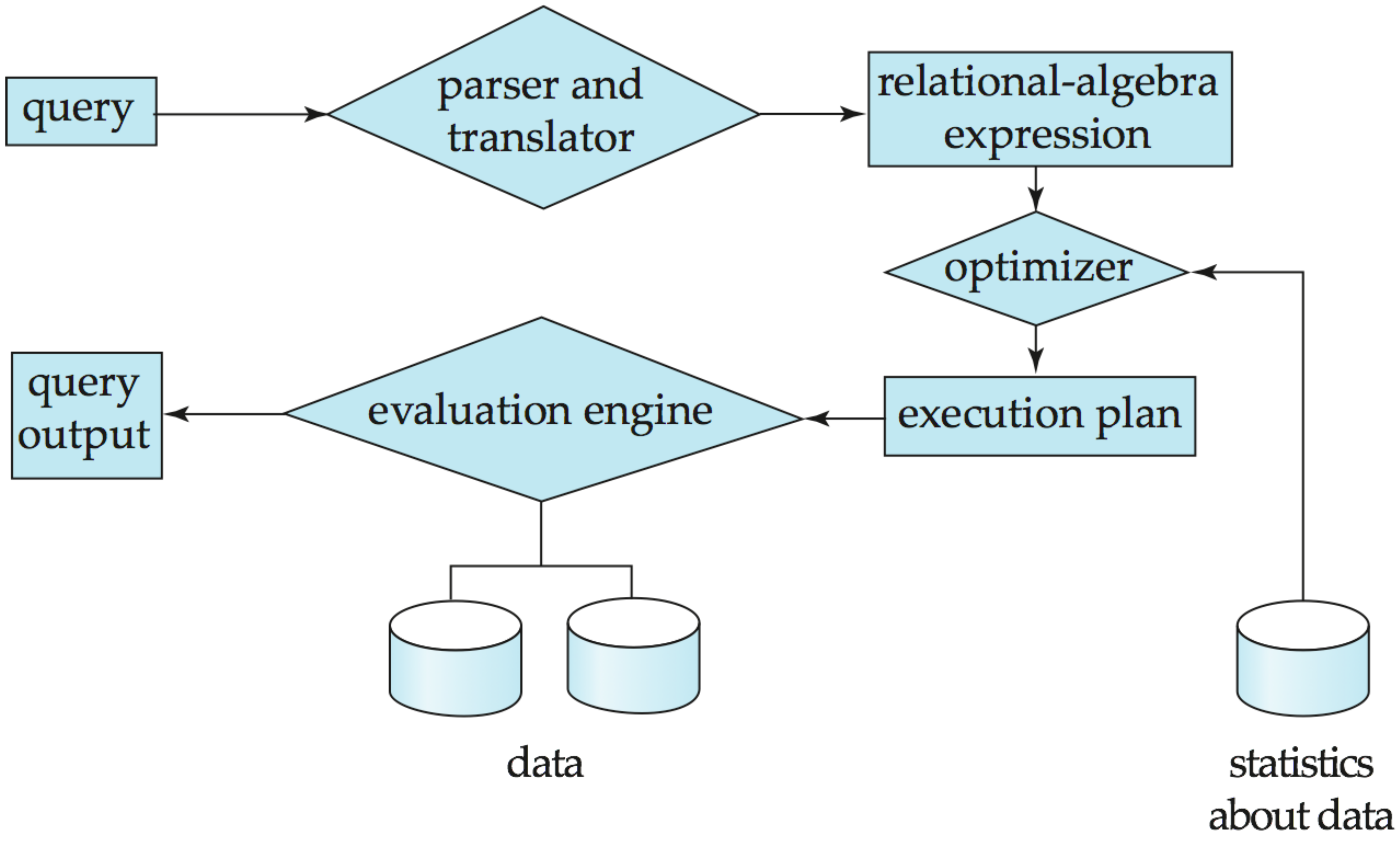
- Suppose we just use a single file or a set of files to store data
- Now, we write programs to use that data
- What are the potential issues?
  - Duplicated work
  - Changes to data layout (schema) require changes to programs
  - New operations required more code
  - Multiple users/programs accessing same data?
  - Security

Database Management System (DBMS)

- Software to manage databases
- Instead of each program writing its own methods to manage data, abstract data management to the DBMS
- Provide levels of abstraction
  - Physical: storage
  - Logical: structure (records, columns, etc.)
  - View: queries and application-support
- Goal: general-purpose
  - Specify structure of the data (schema)
  - Provide query capabilities

Query Processing

- Parsing and translation
- Optimization
- Evaluation



[A. Silberschatz et al.]

Types of Databases

- Many kinds of databases, based on usage
- Amount of data being managed
  - embedded databases: small, application-specific (e.g. SQLite, BerkeleyDB)
  - data warehousing: vast quantities of data (e.g. Oracle)
- Type/frequency of operations being performed
  - OLTP: Online Transaction Processing (e.g. online shopping)
  - OLAP: Online Analytical Processing (e.g. sales analysis)

[D. Pinkston]

Data Models

- Databases must represent:
  - the data itself (typically structured in some way)
  - associations between different data values
  - optionally, constraints on data values
- What kind of data/associations can be represented?
- The data model specifies:
  - what data can be stored (and sometimes how it is stored)
  - associations between different data values
  - what constraints can be enforced
  - how to access and manipulate the data

[D. Pinkston]

Different Data Models

- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semistructured data model (XML)
- Other older models:
  - Network model
  - Hierarchical model

[A. Silberschatz et al.]

Relational Model History

- Invented by Edgar F. Codd in early 1970s
- Focus was data independence
  - Previous data models required physical-level design and implementation
  - Changes to a database schema were very costly to applications that accessed the database
- IBM, Oracle were first implementers of relational model (1977)
  - Usage spread very rapidly through software industry
  - SQL was a particularly powerful innovation

[D. Pinkston]

Relations

- Relations are basically tables of data
  - Each row represents a **tuple** in the relation
- A relational database is an **unordered** set of relations
  - Each relation has a unique name in the database
- Each row in the table specifies a relationship between the values in that row
  - The account ID "A-307", branch name "Seattle", and balance "275" are all related to each other

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

[D. Pinkston]

Relations and Attributes

- Each relation has some number of **attributes**
  - Sometimes called "columns"
- Each attribute has a **domain**
  - Set of valid values for the attribute (+ `null`)
  - Values are usually **atomic**
- The `account` relation has 3 attributes
  - Domain of `balance` is the set of nonnegative integers
  - Domain of `branch_name` is the set of all valid branch names in the bank

acct_id	branch_name	balance
A-301	New York	350
A-307	Seattle	275
A-318	Los Angeles	550
…	…	…

[D. Pinkston]

Database Schema

- Database schema: the logical structure of the database.
- Database instance: a snapshot of the data at a given instant in time.
- Example Schema
  - `instructor` (*ID*, *name*, *dept_name*, *salary*)

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

[A. Silberschatz et al.]