Advanced Data Management (CSCI 680/490)

Databases

Dr. David Koop
Python Containers

• Container: store more than one value
• Mutable versus immutable: Can we update the container?
  - Yes $\rightarrow$ mutable
  - No $\rightarrow$ 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:
    \[<\text{start}>:<\text{end}>\]
  - 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, \( \text{set}() \) is an empty set
- Adding values: \( s.\text{add}(\text{"ME"}) \)
- Removing values: \( s.\text{discard}(\text{"CT"}) \)
- Exists: "CT" in \( s \)
- Union: \( s \mid t \Rightarrow \{ 'MA', 'RI', 'CT', 'NH', 'NY' \} \)
- Intersection: \( s \& t \Rightarrow \{ 'MA', 'NH' \} \)
- Exclusive-or (xor): \( s \uparrow t \Rightarrow \{ 'RI', 'CT', 'NY' \} \)
- Difference: \( s - t \Rightarrow \{ 'RI', 'CT' \} \)
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
  
  ```python
  import collections
d = collections.defaultdict(list)
d[3].append(1)
  ```

- import <name> as <shorter-name>
  
  ```python
  import collections as c
  ```

- from <module> import <name> – don't need to refer to the module
  
  ```python
  from collections import defaultdict
d = defaultdict(list)
d[3].append(1)
  ```
Other Collections Features

- `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
Assignment 1

• Due Monday, Feb. 7 at 11:59pm
• Using Python for data analysis on the Met's artwork
• Provided a1.ipynb file (right-click and download)
• Use basic python for now to demonstrate language knowledge
  - No pandas (for now)
• Use Anaconda or hosted Python environment
• Turn .ipynb file in via Blackboard
• Notes:
  - You will need to do some parsing of the data (converting to ints, splitting strings)
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
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)
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
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
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
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

<table>
<thead>
<tr>
<th>acct_id</th>
<th>branch_name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-301</td>
<td>New York</td>
<td>350</td>
</tr>
<tr>
<td>A-307</td>
<td>Seattle</td>
<td>275</td>
</tr>
<tr>
<td>A-318</td>
<td>Los Angeles</td>
<td>550</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>acct_id</th>
<th>branch_name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-301</td>
<td>New York</td>
<td>350</td>
</tr>
<tr>
<td>A-307</td>
<td>Seattle</td>
<td>275</td>
</tr>
<tr>
<td>A-318</td>
<td>Los Angeles</td>
<td>550</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
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)

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

[A. Silberschatz et al.]
Keys

- Let $K \subseteq R$
- $K$ is a **superkey** of $R$ if values for $K$ are sufficient to identify a unique tuple of each possible relation $r(R)$
  - Example: \{ID\} and \{ID, name\} are both superkeys of instructor.
- Superkey $K$ is a **candidate key** if $K$ is **minimal**
  - Example: \{ID\} is a candidate key for Instructor
- One of the candidate keys is selected to be the **primary key**.
  - Which one?
Foreign Key Constraints

• Foreign key constraint: Value in one relation **must appear** in another
  - *Referencing* relation
  - *Referenced* relation
  - Example: `dept_name` in `instructor` is a foreign key from `instructor` referencing `department`
Relational Query Languages

• Procedural versus non-procedural, or declarative
• “Pure” languages:
  - Relational algebra
  - Tuple relational calculus
  - Domain relational calculus
• The above 3 pure languages are equivalent in computing power
• Concentrate on relational algebra
  - Not Turing-machine equivalent
  - 6 basic operations
Relational Algebra

• Definition: A procedural language consisting of a set of operations that take one or two relations as input and produce a new relation as their result.

• Six basic operators
  - select: $\sigma$
  - project: $\Pi$
  - union: $\cup$
  - set difference: $-$
  - Cartesian product: $\times$
  - rename: $\rho$
Select Operation

- The select operation selects tuples that satisfy a given predicate.
- Notation: $\sigma_p(r)$
- $p$ is called the selection predicate
- Example: select those tuples of the instructor relation where the instructor is in the “Physics” department.
  - Query: $\sigma_{\text{dept\_name}=\text{“Physics”}}(\text{instructor})$
  - Result:

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>Physics</td>
<td>87000</td>
</tr>
</tbody>
</table>
Select Operation Comparisons

• We allow comparisons using $=$, $\neq$, $>$, $\geq$, $<$, $\leq$ in the selection predicate.

• We can combine several predicates into a larger predicate by using the connectives: $\land$ (and), $\lor$ (or), $\neg$ (not)

• Example: Find the instructors in Physics with a salary greater than $90,000$:
  - $\sigma_{\text{dept\_name}=\text{"Physics"}} \land \text{salary} > 90,000 \ (\text{instructor})$

• The select predicate may include comparisons between two attributes.
  - Example: departments whose name is the same as their building name:
    - $\sigma_{\text{dept\_name}=\text{building}} \ (\text{department})$
Project Operation

- A unary operation that returns its argument relation, with certain attributes left out.
- Notation: \( \prod_{A_1, A_2, A_3, \ldots, A_k} (r) \)
  where \( A_1, A_2, A_3, \ldots, A_k \) are attribute names and \( r \) is a relation name.
- The result is defined as the relation of \( k \) columns obtained by erasing the columns that are not listed.
- Duplicate rows removed from result, since relations are sets.
Project Operation Example

- **Example:** eliminate the `dept_name` attribute of instructor
- **Query:** \( \prod_{ID, \text{name}, \text{salary}}(\text{instructor}) \)

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>65000</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>90000</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>40000</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>95000</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>60000</td>
</tr>
<tr>
<td>33456</td>
<td>Gold</td>
<td>87000</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>75000</td>
</tr>
<tr>
<td>58583</td>
<td>Califieri</td>
<td>62000</td>
</tr>
<tr>
<td>76543</td>
<td>Singh</td>
<td>80000</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>72000</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>92000</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>80000</td>
</tr>
</tbody>
</table>
Composition of Relational Operations

- The result of a relational-algebra operation is a relation
- … so relational-algebra operations can be composed together into a relational-algebra expression.
- Example: Find the names of all instructors in the Physics department.

\[ \Pi_{\text{name}}(\sigma_{\text{dept_name} = \text{"Physics"}}(\text{instructor})) \]

- Instead of giving the name of a relation as the argument of the projection operation, we give an expression that evaluates to a relation.
Cartesian-Product Operation

- The **Cartesian-product** operation (denoted by X) allows us to combine information from any two relations.

- Example: the Cartesian product of the relations *instructor* and *teaches* is written as: instructor X teaches

- We construct a tuple of the result out of each possible pair of tuples: one from the instructor relation and one from the teaches relation.

- Since the instructor ID appears in both relations we distinguish between these attribute by attaching to the attribute the name of the relation from which the attribute originally came.
  - instructor.ID and teaches.ID
The instructor X teaches table

<table>
<thead>
<tr>
<th>instructor.ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
<th>teaches.ID</th>
<th>course_id</th>
<th>sec_id</th>
<th>semester</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>12121</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>12121</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
</tbody>
</table>

[A.: Silberschatz et al.]
Join Operation

• The Cartesian-Product $instructor \times teaches$ associates every tuple of instructor with every tuple of teaches.
  - Most of the resulting rows have information about instructors who did not teach a particular course.

• To get only those tuples of $instructor \times teaches$ that pertain to instructors and the courses that they taught, we write:

  $$\sigma_{instructor.id = teaches.id} (instructor \times teaches)$$

  - We get only those tuples of $instructor \times teaches$ that pertain to instructors and the courses that they taught.
Join Operation (Cont.)

The table corresponding to $\sigma_{\text{instructor}.id = \text{teaches}.id} (\text{instructor} \times \text{teaches})$

<table>
<thead>
<tr>
<th>instructor.ID</th>
<th>name</th>
<th>dept_name</th>
<th>salary</th>
<th>teaches.ID</th>
<th>course_id</th>
<th>sec_id</th>
<th>semester</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>10101</td>
<td>Srinivasan</td>
<td>Comp. Sci.</td>
<td>65000</td>
<td>10101</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>12121</td>
<td>Wu</td>
<td>Finance</td>
<td>90000</td>
<td>12121</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>15151</td>
<td>Mozart</td>
<td>Music</td>
<td>40000</td>
<td>15151</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>22222</td>
<td>Einstein</td>
<td>Physics</td>
<td>95000</td>
<td>22222</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
</tr>
<tr>
<td>32343</td>
<td>El Said</td>
<td>History</td>
<td>60000</td>
<td>32343</td>
<td>HIS-351</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
<td>45565</td>
<td>CS-101</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>45565</td>
<td>Katz</td>
<td>Comp. Sci.</td>
<td>75000</td>
<td>45565</td>
<td>CS-319</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
<td>76766</td>
<td>BIO-101</td>
<td>1</td>
<td>Summer</td>
<td>2017</td>
</tr>
<tr>
<td>76766</td>
<td>Crick</td>
<td>Biology</td>
<td>72000</td>
<td>76766</td>
<td>BIO-301</td>
<td>1</td>
<td>Summer</td>
<td>2018</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
<td>83821</td>
<td>CS-190</td>
<td>1</td>
<td>Spring</td>
<td>2017</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
<td>83821</td>
<td>CS-190</td>
<td>2</td>
<td>Spring</td>
<td>2017</td>
</tr>
<tr>
<td>83821</td>
<td>Brandt</td>
<td>Comp. Sci.</td>
<td>92000</td>
<td>83821</td>
<td>CS-319</td>
<td>2</td>
<td>Spring</td>
<td>2018</td>
</tr>
<tr>
<td>98345</td>
<td>Kim</td>
<td>Elec. Eng.</td>
<td>80000</td>
<td>98345</td>
<td>EE-181</td>
<td>1</td>
<td>Spring</td>
<td>2017</td>
</tr>
</tbody>
</table>
Join Operation

• The **join** operation allows us to combine a select operation and a Cartesian-Product operation into a single operation.

• Consider relations $r(R)$ and $s(S)$

• Let $\theta$ be a predicate on attributes in the schema $R \cup S$. The join operation is:

$$ r \bowtie_{\theta} s = \sigma_{\theta} (r \times s) $$

• Thus

$$ \sigma_{instructor.id = teaches.id} (instructor \times teaches) $$

• can equivalently be written as

$$ instructor \bowtie_{Instructor.id = teaches.id} teaches $$
Union Operation

• The **union** operation allows us to combine two relations
• Notation: \( r \cup s \)
• For \( r \cup s \) to be valid.
   - \( r, s \) must have the same arity (same number of **attributes**)
   - The attribute domains must be **compatible** (example: 2nd column of \( r \) deals with the same type of values as does the 2nd column of \( s \))
Union Example

• Find all courses taught in the Fall 2017 semester, or in the Spring 2018 semester, or in both:

$\bigcap_{\text{course}_i} (\sigma_{\text{semester}="Fall" \land \text{year}=2017} (\text{section})) \cup \bigcap_{\text{course}_i} (\sigma_{\text{semester}="Spring" \land \text{year}=2018} (\text{section}))$

<table>
<thead>
<tr>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
</tr>
<tr>
<td>CS-319</td>
</tr>
<tr>
<td>CS-347</td>
</tr>
<tr>
<td>FIN-201</td>
</tr>
<tr>
<td>HIS-351</td>
</tr>
<tr>
<td>MU-199</td>
</tr>
<tr>
<td>PHY-101</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
Set-Intersection Operation

• The **set-intersection** operation allows us to find tuples that are in both the input relations.

• Notation: \( r \cap s \)

• Same requirements as union:
  - \( r, s \) have the same arity
  - attributes of \( r \) and \( s \) are compatible

• Example: Find the set of all courses taught in both the Fall 2017 and the Spring 2018 semesters.

\[ \prod_{\text{course_id}} (\sigma_{\text{semester}="\text{Fall}" \land \text{year}=2017} (\text{section})) \cap \prod_{\text{course_id}} (\sigma_{\text{semester}="\text{Spring}" \land \text{year}=2018} (\text{section})) \]

[A. Silberschatz et al.]
Set Difference Operation

- The **set-difference** operation allows us to find tuples that are in one relation but are not in another.
- Notation $r - s$
- Same requirements as union and set-intersection: .
  - $r$ and $s$ must have the same arity
  - attribute domains of $r$ and $s$ must be compatible
- Example: Find all courses taught in the Fall 2017 semester, but not in the Spring 2018 semester

\[
\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Fall"}} \land \text{year}=2017 (\text{section})) - \\
\Pi_{\text{course_id}} (\sigma_{\text{semester}=\text{"Spring"}} \land \text{year}=2018 (\text{section}))
\]

<table>
<thead>
<tr>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS-347</td>
</tr>
<tr>
<td>PHY-101</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
Equivalent Queries

• There is more than one way to write a query in relational algebra.
• Example: Find information about courses taught by instructors in the Physics department with salary greater than 90,000
• Query 1: $\sigma_{\text{dept\_name}=\text{"Physics"}} \land \text{salary} > 90,000 (\text{instructor})$
• Query 2: $\sigma_{\text{dept\_name}=\text{"Physics"}} (\sigma_{\text{salary} > 90.000} (\text{instructor}))$
• The two queries are not identical; they are, however, equivalent -- they give the same result on any database.
Equivalent Queries

• Example: Find information about courses taught by instructors in the Physics department

• Query 1:

\[ \sigma_{\text{dept}_\text{name}="\text{Physics}"}(\text{instructor} \bowtie \text{instructor.}\text{ID} = \text{teaches.}\text{ID} \text{ teaches}) \]

• Query 2

\[ (\sigma_{\text{dept}_\text{name}="\text{Physics}"}(\text{instructor})) \bowtie \text{instructor.}\text{ID} = \text{teaches.}\text{ID} \text{ teaches} \]

• The order of joins is one focus of some of the work on query optimization