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 → 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'}`

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 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

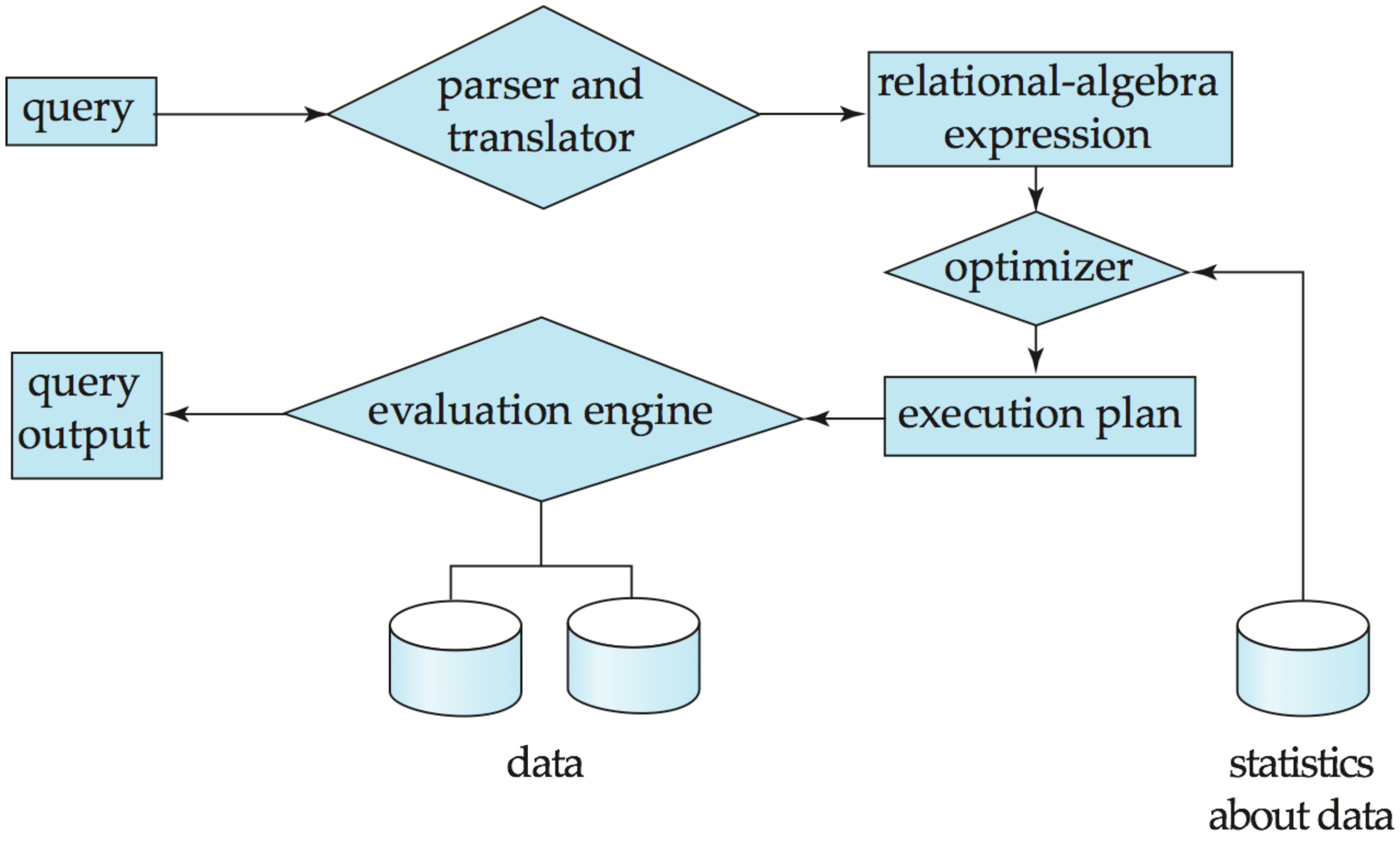
- Suppose we just use a single file or a set of files to store data
- Now, we write programs to use that data
- What are the potential issues?
  - Duplicated work
  - Changes to data layout (schema) require changes to programs
  - New operations required more code
  - Multiple users/programs accessing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



[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.]

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?

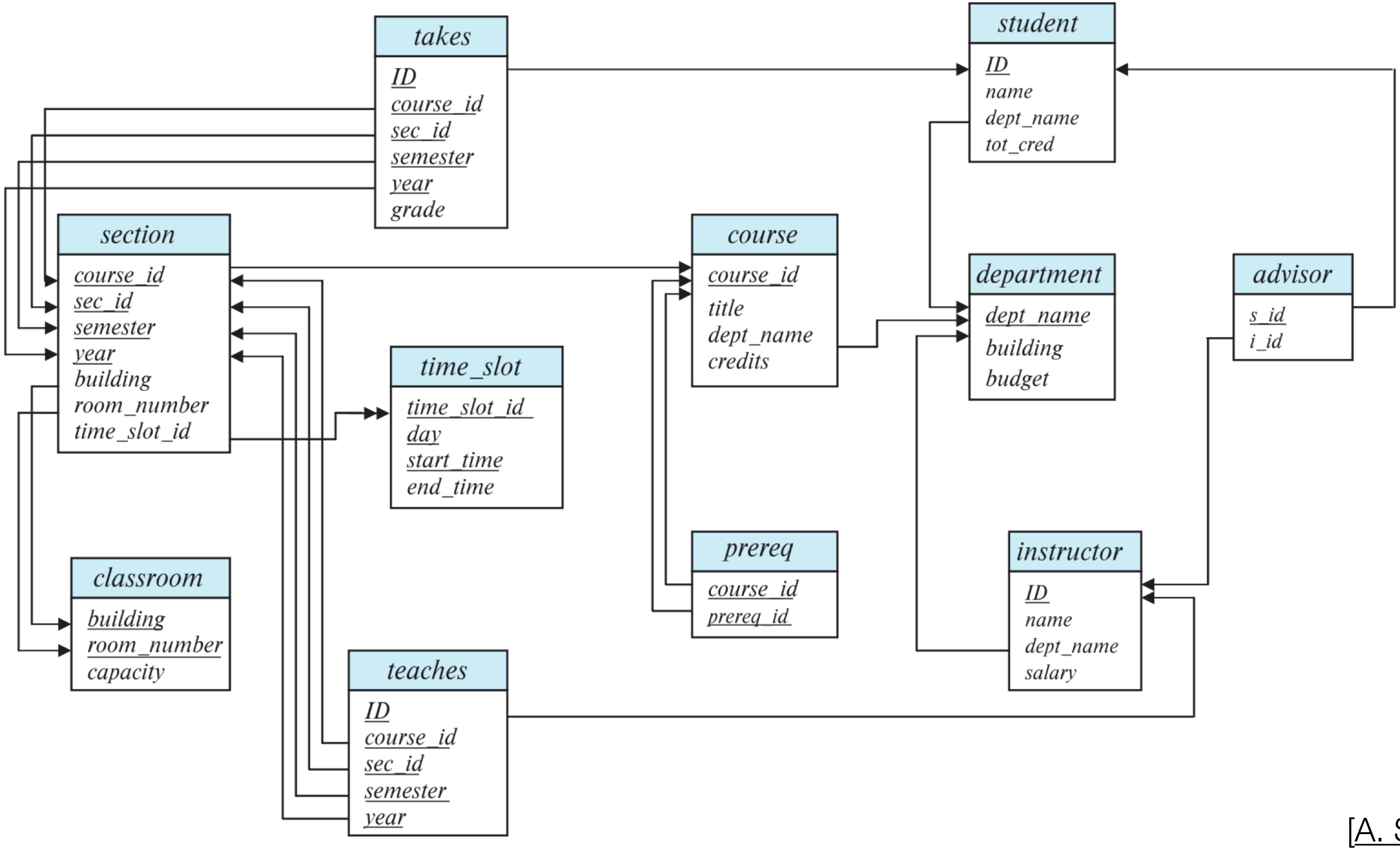
[A. Silberschatz et al.]

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`

[A. Silberschatz et al.]

Schema Diagram with Keys



[A. Silberschatz et al.]

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

[A. Silberschatz et al.]

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: σ
  - project: ∏
  - union: ∪
  - set difference: –
  - Cartesian product: x
  - rename: ρ

[A. Silberschatz et al.]

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: $\sigma_{dept\_name=\text{“Physics”}}$(`instructor`)
  - Result:

ID	*name*	*dept_name*	*salary*
22222	Einstein	Physics	95000
33456	Gold	Physics	87000

[A. Silberschatz et al.]

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: ∧ (and), ∨ (or), ¬ (not)
- Example: Find the instructors in Physics with a salary greater than $90,000:
  - $\sigma_{\text{dept\_name}=\text{“Physics”} \wedge \text{salary} > 90{,}000}(\texttt{instructor})$
-
- The select predicate may include comparisons between two **attributes**.
  - Example: departments whose name is the same as their building name:
    - $\sigma_{\text{dept\_name}=\text{building}}(\texttt{department})$

[A. Silberschatz et al.]

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

[A. Silberschatz et al.]

Project Operation Example

ID	*name*	*salary*
10101	Srinivasan	65000
12121	Wu	90000
15151	Mozart	40000
22222	Einstein	95000
32343	El Said	60000
33456	Gold	87000
45565	Katz	75000
58583	Califieri	62000
76543	Singh	80000
76766	Crick	72000
83821	Brandt	92000
98345	Kim	80000

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

[A. Silberschatz et al.]

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.

$$\prod_{\text{name}}(\sigma_{\text{dept\_name} = \text{“Physics”}} (\texttt{instructor}))$$

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

[A. Silberschatz et al.]

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`

[A. Silberschatz et al.]

The instructor X teaches table

instructor.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2017
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2018
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2017
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2018
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2017
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2018
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2018
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2017
...	...	...	...	...	...	...	...	...

[A. Silberschatz et al.]

Join Operation

- The Cartesian-Product `instructor X 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 X teaches` that pertain to instructors and the courses that they taught, we write:

$$\sigma_{\text{instructor.id = teaches.id}}\ (\texttt{instructor x teaches})$$

  - We get only those tuples of `instructor X teaches` that pertain to instructors and the courses that they taught.

[A. Silberschatz et al.]

Join Operation (Cont.)

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

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

[A. Silberschatz et al.]

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 θ be a predicate on attributes in the schema R ∪ S. The join operation is:

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

- Thus

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

- can equivalently be written as

$$\texttt{instructor} \bowtie_{\text{Instructor.id = teaches.id}} \texttt{teaches}$$

[A. Silberschatz et al.]

Union Operation

- The **union** operation allows us to combine two relations
- Notation: `r ∪ s`
- For `r ∪ 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`)

[A. Silberschatz et al.]

Union Example

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

$$\prod_{course\_id} (\sigma_{semester=\text{“Fall”} \land year=2017} (\texttt{section})) \cup$$
$$\prod_{course\_id} (\sigma_{semester=\text{“Spring”} \land year=2018} (\texttt{section}))$$

course_id
CS-101
CS-315
CS-319
CS-347
FIN-201
HIS-351
MU-199
PHY-101

[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 ∩ 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.
- $\prod_{\text{course\_id}} (\sigma_{\text{semester=“Fall”} \land \text{year=2017}} (\texttt{section})) \cap$

$\prod_{\text{course\_id}} (\sigma_{\text{semester=“Spring”} \land \text{year=2018}} (\texttt{section}))$

course_id
CS-101

[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

$$\prod_{\text{course\_id}} (\sigma_{\text{semester="Fall"} \wedge \text{year=2017}} (\texttt{section})) - \prod_{\text{course\_id}} (\sigma_{\text{semester="Spring"} \wedge \text{year=2018}} (\texttt{section}))$$

course_id
CS-347
PHY-101

[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: $\sigma_{\text{dept\_name="Physics"} \wedge \text{ salary} > 90{,}000}$ (`instructor`)
- Query 2: $\sigma_{\text{dept\_name="Physics"}}$ ($\sigma_{\text{salary} > 90.000}$ (`instructor`))
- The two queries are **not identical**; they are, however, **equivalent** -- they give the same result on any database.

[A. Silberschatz et al.]

Equivalent Queries

- Example: Find information about courses taught by instructors in the Physics department
- Query 1:

$$\sigma_{\text{dept\_name=“Physics”}} (\texttt{instructor} \bowtie_{\text{instructor.ID = teaches.ID}} \texttt{teaches})$$

- Query 2

$$(\sigma_{\text{dept\_name=“Physics”}} (\texttt{instructor})) \bowtie_{\text{instructor.ID = teaches.ID}} \texttt{teaches}$$

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

[A. Silberschatz et al.]