Advanced Data Management (CSCI 640/490)

Structured Data

Dr. David Koop
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: $\rho$

[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 \textit{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>
Project Operation

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

<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>
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 attributes by attaching to the attribute the name of the relation from which the attribute originally came.

- `instructor.ID` and `teaches.ID`
Join Operation

- The Cartesian-Product $\text{instructor} \times \text{teaches}$ 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 $\text{instructor} \times \text{teaches}$ that pertain to instructors and the courses that they taught, we write:
  \[ \sigma_{\text{instructor}.id = \text{teaches}.id} (\text{instructor} \times \text{teaches}) \]
  - We get only those tuples of $\text{instructor} \times \text{teaches}$ that pertain to instructors and the courses that they taught.

[A. Silberschatz et al.]
Equivalent Queries

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

- Query 1:
  \[ \sigma_{\text{dept}_\text{name}=“\text{Physics}”} (\text{instructor} \bowtie \text{instructor.ID} = \text{teaches.ID} \text{ teaches}) \]

- Query 2
  \[ (\sigma_{\text{dept}_\text{name}=“\text{Physics}”} (\text{instructor})) \bowtie \text{instructor.ID} = \text{teaches.ID} \text{ teaches} \]

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

• **Data Definition Language (DDL):** the specification of information about relations, including schema, types, integrity constraints, indices, storage

• **Data Manipulation Language (DML):** provides the ability to query information from the database and to insert tuples into, delete tuples from, and modify tuples in the database.

• **Integrity:** the DDL includes commands for specifying integrity constraints.

• **View definition:** The DDL includes commands for defining views.

• Also: **Transaction control, embedded and dynamic SQL, authorization**
Create Table

- An SQL relation is defined using the create table command:

\[
\text{create table } r (A_1 D_1, A_2 D_2, \ldots, A_n D_n, (C_1), \ldots, (C_k))
\]

- \(r\) is the name of the relation
- each \(A_i\) is an attribute name in the schema of relation \(r\)
- \(D_i\) is the data type of values in the domain of attribute \(A_i\)

- Example:

\[
\text{create table instructor(}
\begin{align*}
\text{ID} & \quad \text{char(5),} \\
\text{name} & \quad \text{varchar(20),} \\
\text{dept\_name} & \quad \text{varchar(20),} \\
\text{salary} & \quad \text{numeric(8,2));}
\end{align*}
\]

\(C_i\) are integrity constraints: keys, foreign keys
Basic Query Structure

• A typical SQL query has the form:

\[
\text{select } A_1, A_2, \ldots, A_n \\
\text{from } r_1, r_2, \ldots, r_m \\
\text{where } P
\]

- \( A_i \) represents an attribute
- \( r_i \) represents a relation
- \( P \) is a predicate.

• The result of an SQL query is a relation
Select

- The **select** clause lists the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: Find the names of all instructors
  - **select** name
    from instructor;
- Note: SQL names are **case insensitive**
  - Name and NAME and name are equivalent
  - Some people use upper case for language keywords (e.g. **SELECT**)
Where

- The operands can be expressions with operators $<$, $<=$, $>$, $=>$, $=$, and $<>$
- SQL allows the use of the logical connectives `and`, `or`, and `not`
- Comparisons can be applied to results of arithmetic expressions
- Example: Find all instructors in Comp. Sci. with salary $>$ 70000
  
  ```sql
  select name
  from instructor
  where dept_name = 'Comp. Sci.' and salary > 70000
  ```
From

- The **from** clause lists the relations involved in the query
  - Corresponds to the **Cartesian Product** operation in relational algebra
- Find the Cartesian product **instructor X teaches**
  - **select** *
    - **from** instructor, teaches;
  - All possible instructor – teaches pair, with all attributes from both
  - Shared attributes (e.g., ID) are renamed (e.g., instructor.ID)
- Not very useful directly but useful combined with where clauses.

[A. Silberschatz et al.]

D. Koop, CSCI 640/490, Spring 2023
Group By

- Find the average salary of instructors in each department

```sql
select dept_name, avg(salary) as avg_salary 
from instructor 
group by dept_name;
```

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

<table>
<thead>
<tr>
<th>dept_name</th>
<th>avg_salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Biology</td>
<td>72000</td>
</tr>
<tr>
<td>Comp. Sci.</td>
<td>77333</td>
</tr>
<tr>
<td>Elec. Eng.</td>
<td>80000</td>
</tr>
<tr>
<td>Finance</td>
<td>85000</td>
</tr>
<tr>
<td>History</td>
<td>61000</td>
</tr>
<tr>
<td>Music</td>
<td>40000</td>
</tr>
<tr>
<td>Physics</td>
<td>91000</td>
</tr>
</tbody>
</table>
Deletion

- Delete all instructors: `delete from instructor;`
- Delete all instructors from the Finance department
  - `delete from instructor
    where dept_name= 'Finance';`
- Delete all tuples in the instructor relation for those instructors associated with a department located in the Watson building
  - `delete from instructor
    where dept_name in (select dept_name
                        from department
                        where building = 'Watson');`
Insertion

• Make each student in the Music department who has earned more than 144 credit hours an instructor in the Music department with a salary of $18,000.

- `insert into instructor
  select ID, name, dept_name, 18000
  from student
  where dept_name = 'Music' and total_cred > 144;`

• The select-from-where statement is evaluated fully before any of its results are inserted into the relation.

• If not queries like

  `insert into table1 select * from table1`

would cause problems

[A. Silberschatz et al.]
Updates

• Give a 5% salary raise to all instructors
  - `update instructor
    set salary = salary * 1.05`

• Give a 5% salary raise to those instructors who earn less than 70000
  - `update instructor
    set salary = salary * 1.05
    where salary < 70000;

• Give a 5% salary raise to instructors whose salary is less than average
  - `update instructor
    set salary = salary * 1.05
    where salary < (select avg(salary) from instructor);`
Assignment 2

• Same questions as Assignment 1 but using pandas, duckdb, and ibis
Joins

• Join operations take two relations and return another relation.
• From relational algebra, this is a Cartesian product + selection
• Want tuples in the two relations to match (under some condition)
• The join operations typically used as subquery expressions in the from clause
• Three types of joins:
  - Natural join
  - Inner join
  - Outer join
Natural Join

• Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column.

• List the names of instructors along with the course ID of the courses that they taught

  - `select name, course_id`
    `from students, takes`
    `where student.ID = takes.ID;`

• Same query in SQL with “natural join” construct

  - `select name, course_id`
    `from student natural join takes;`
## Example: Student Schedules

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>tot_cred</th>
</tr>
</thead>
<tbody>
<tr>
<td>00128</td>
<td>Zhang</td>
<td>Comp. Sci.</td>
<td>102</td>
</tr>
<tr>
<td>12345</td>
<td>Shankar</td>
<td>Comp. Sci.</td>
<td>32</td>
</tr>
<tr>
<td>19991</td>
<td>Brandt</td>
<td>History</td>
<td>80</td>
</tr>
<tr>
<td>23121</td>
<td>Chavez</td>
<td>Finance</td>
<td>110</td>
</tr>
<tr>
<td>44553</td>
<td>Peltier</td>
<td>Physics</td>
<td>56</td>
</tr>
<tr>
<td>45678</td>
<td>Levy</td>
<td>Physics</td>
<td>46</td>
</tr>
<tr>
<td>54321</td>
<td>Williams</td>
<td>Comp. Sci.</td>
<td>54</td>
</tr>
<tr>
<td>55739</td>
<td>Sanchez</td>
<td>Music</td>
<td>38</td>
</tr>
<tr>
<td>70557</td>
<td>Snow</td>
<td></td>
<td>0</td>
</tr>
<tr>
<td>76543</td>
<td>Brown</td>
<td>Comp. Sci.</td>
<td>58</td>
</tr>
<tr>
<td>76653</td>
<td>Aoi</td>
<td>Elec. Eng.</td>
<td>60</td>
</tr>
<tr>
<td>98765</td>
<td>Bourikas</td>
<td>Elec. Eng.</td>
<td>98</td>
</tr>
<tr>
<td>98988</td>
<td>Tanaka</td>
<td>Biology</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>course_id</th>
<th>sec_id</th>
<th>semester</th>
<th>year</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>00128</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>00128</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A-</td>
</tr>
<tr>
<td>12345</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>C</td>
</tr>
<tr>
<td>12345</td>
<td>CS-190</td>
<td>2</td>
<td>Spring</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>12345</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>A</td>
</tr>
<tr>
<td>12345</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>19991</td>
<td>HIS-351</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B</td>
</tr>
<tr>
<td>23121</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>C+</td>
</tr>
<tr>
<td>44553</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>B-</td>
</tr>
<tr>
<td>45678</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>F</td>
</tr>
<tr>
<td>45678</td>
<td>CS-101</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B+</td>
</tr>
<tr>
<td>45678</td>
<td>CS-319</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B</td>
</tr>
<tr>
<td>54321</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A-</td>
</tr>
<tr>
<td>54321</td>
<td>CS-190</td>
<td>2</td>
<td>Spring</td>
<td>2017</td>
<td>B+</td>
</tr>
<tr>
<td>55739</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>A-</td>
</tr>
<tr>
<td>76543</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>76543</td>
<td>CS-319</td>
<td>2</td>
<td>Spring</td>
<td>2018</td>
<td>A</td>
</tr>
<tr>
<td>76653</td>
<td>EE-181</td>
<td>1</td>
<td>Spring</td>
<td>2017</td>
<td>C</td>
</tr>
<tr>
<td>98765</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>C-</td>
</tr>
<tr>
<td>98765</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B</td>
</tr>
<tr>
<td>98988</td>
<td>BIO-101</td>
<td>1</td>
<td>Summer</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>98988</td>
<td>BIO-301</td>
<td>1</td>
<td>Summer</td>
<td>2018</td>
<td>null</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
## Example: Natural Join

<table>
<thead>
<tr>
<th>ID</th>
<th>name</th>
<th>dept_name</th>
<th>tot_cred</th>
<th>course_id</th>
<th>sec_id</th>
<th>semester</th>
<th>year</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>00128</td>
<td>Zhang</td>
<td>Comp. Sci.</td>
<td>102</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>00128</td>
<td>Zhang</td>
<td>Comp. Sci.</td>
<td>102</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A-</td>
</tr>
<tr>
<td>12345</td>
<td>Shankar</td>
<td>Comp. Sci.</td>
<td>32</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>C</td>
</tr>
<tr>
<td>12345</td>
<td>Shankar</td>
<td>Comp. Sci.</td>
<td>32</td>
<td>CS-190</td>
<td>2</td>
<td>Spring</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>12345</td>
<td>Shankar</td>
<td>Comp. Sci.</td>
<td>32</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>A</td>
</tr>
<tr>
<td>12345</td>
<td>Shankar</td>
<td>Comp. Sci.</td>
<td>32</td>
<td>CS-347</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>19991</td>
<td>Brandt</td>
<td>History</td>
<td>80</td>
<td>HIS-351</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B</td>
</tr>
<tr>
<td>23121</td>
<td>Chavez</td>
<td>Finance</td>
<td>110</td>
<td>FIN-201</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>C+</td>
</tr>
<tr>
<td>44553</td>
<td>Peltier</td>
<td>Physics</td>
<td>56</td>
<td>PHY-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>B</td>
</tr>
<tr>
<td>45678</td>
<td>Levy</td>
<td>Physics</td>
<td>46</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>F</td>
</tr>
<tr>
<td>45678</td>
<td>Levy</td>
<td>Physics</td>
<td>46</td>
<td>CS-101</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B+</td>
</tr>
<tr>
<td>45678</td>
<td>Levy</td>
<td>Physics</td>
<td>46</td>
<td>CS-319</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B</td>
</tr>
<tr>
<td>54321</td>
<td>Williams</td>
<td>Comp. Sci.</td>
<td>54</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A-</td>
</tr>
<tr>
<td>54321</td>
<td>Williams</td>
<td>Comp. Sci.</td>
<td>54</td>
<td>CS-190</td>
<td>2</td>
<td>Spring</td>
<td>2017</td>
<td>B+</td>
</tr>
<tr>
<td>55739</td>
<td>Sanchez</td>
<td>Music</td>
<td>38</td>
<td>MU-199</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>A-</td>
</tr>
<tr>
<td>76543</td>
<td>Brown</td>
<td>Comp. Sci.</td>
<td>58</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>76543</td>
<td>Brown</td>
<td>Comp. Sci.</td>
<td>58</td>
<td>CS-319</td>
<td>2</td>
<td>Spring</td>
<td>2018</td>
<td>A</td>
</tr>
<tr>
<td>76653</td>
<td>Aoi</td>
<td>Elec. Eng.</td>
<td>60</td>
<td>EE-181</td>
<td>1</td>
<td>Spring</td>
<td>2017</td>
<td>C</td>
</tr>
<tr>
<td>98765</td>
<td>Bourikas</td>
<td>Elec. Eng.</td>
<td>98</td>
<td>CS-101</td>
<td>1</td>
<td>Fall</td>
<td>2017</td>
<td>C-</td>
</tr>
<tr>
<td>98765</td>
<td>Bourikas</td>
<td>Elec. Eng.</td>
<td>98</td>
<td>CS-315</td>
<td>1</td>
<td>Spring</td>
<td>2018</td>
<td>B</td>
</tr>
<tr>
<td>98988</td>
<td>Tanaka</td>
<td>Biology</td>
<td>120</td>
<td>BIO-101</td>
<td>1</td>
<td>Summer</td>
<td>2017</td>
<td>A</td>
</tr>
<tr>
<td>98988</td>
<td>Tanaka</td>
<td>Biology</td>
<td>120</td>
<td>BIO-301</td>
<td>1</td>
<td>Summer</td>
<td>2018</td>
<td>null</td>
</tr>
</tbody>
</table>
Natural Join Danger

• Beware of unrelated attributes with same name which get equated incorrectly
• Example: List the names of students instructors along with the titles of courses that they have taken
  - `select name, title
   from student natural join takes natural join course;`
• Wrong… only lists courses when the student took courses in their department (major)
• Correct:
  - `select name, title
   from student natural join takes, course
   where takes.course_id = course.course_id;`
Outer Join

• Joins so far are inner joins
• Outer joins returns tuples from one (or both) relations that do not match tuples in the other relation
• Fills in missing values with null
• Three forms of outer join:
  - left outer join
  - right outer join
  - full outer join
Join Examples

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course_id</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

Left Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
</tbody>
</table>

Right Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prereq_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
Join Examples

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>course_id</th>
<th>prerek_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-347</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

(Full) Outer Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prerek_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
</tr>
<tr>
<td>CS-315</td>
<td>Robotics</td>
<td>Comp. Sci.</td>
<td>3</td>
<td>null</td>
</tr>
<tr>
<td>CS-347</td>
<td>null</td>
<td>null</td>
<td>null</td>
<td>CS-101</td>
</tr>
</tbody>
</table>

Inner Join

<table>
<thead>
<tr>
<th>course_id</th>
<th>title</th>
<th>dept_name</th>
<th>credits</th>
<th>prerek_id</th>
<th>course_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIO-301</td>
<td>Genetics</td>
<td>Biology</td>
<td>4</td>
<td>BIO-101</td>
<td>BIO-301</td>
</tr>
<tr>
<td>CS-190</td>
<td>Game Design</td>
<td>Comp. Sci.</td>
<td>4</td>
<td>CS-101</td>
<td>CS-190</td>
</tr>
</tbody>
</table>

[A. Silberschatz et al.]
Arrays

What is the difference between an array and a list (or a tuple)?
Arrays

• Usually a fixed size—lists are meant to change size
• Are mutable—tuples are not
• Store only one type of data—lists and tuples can store anything
• Are faster to access and manipulate than lists or tuples
• Can be multidimensional:
  - Can have list of lists or tuple of tuples but no guarantee on shape
  - Multidimensional arrays are rectangles, cubes, etc.
Why NumPy?

- Fast **vectorized** array operations for data munging and cleaning, subsetting and filtering, transformation, and any other kinds of computations
- Common array algorithms like sorting, unique, and set operations
- Efficient descriptive statistics and aggregating/summarizing data
- Data alignment and relational data manipulations for merging and joining together heterogeneous data sets
- Expressing conditional logic as array expressions instead of loops with `if-elif-else` branches
- Group-wise data manipulations (aggregation, transformation, function application).

[W. McKinney, Python for Data Analysis]
import numpy as np
PyData Notebooks

- ch04.ipynb
- Click the raw button and save that file to disk
- …or download/clone the entire repository
Creating arrays

- data1 = [6, 7, 8, 0, 1]  
  arr1 = np.array(data1)
- data2 = [[1.5, 2, 3, 4], [5, 6, 7, 8]]  
  arr2 = np.array(data2)
- data3 = np.array([6, "abc", 3.57]) # !!! check !!!

- Can check the type of an array in dtype property
- Types:
  - arr1.dtype # dtype('int64')
  - arr3.dtype # dtype('U21'), unicode plus # chars
Types

- "But I thought Python wasn't stingy about types..."
- numpy aims for speed
- Able to do array arithmetic
- int16, int32, int64, float32, float64, bool, object
- Can specify type explicitly
  - `arr1_float = np.array(data1, dtype='float64')`
- `astype` method allows you to convert between different types of arrays:
  ```python
  arr = np.array([1, 2, 3, 4, 5])
  arr.dtype
  float_arr = arr.astype(np.float64)
  ```
### Numpy Data Types (Dtypes)

<table>
<thead>
<tr>
<th>Type</th>
<th>Type Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int8, uint8</td>
<td>i1, u1</td>
<td>Signed and unsigned 8-bit (1 byte) integer types</td>
</tr>
<tr>
<td>int16, uint16</td>
<td>i2, u2</td>
<td>Signed and unsigned 16-bit integer types</td>
</tr>
<tr>
<td>int32, uint32</td>
<td>i4, u4</td>
<td>Signed and unsigned 32-bit integer types</td>
</tr>
<tr>
<td>int64, uint64</td>
<td>i8, u8</td>
<td>Signed and unsigned 64-bit integer types</td>
</tr>
<tr>
<td>float16</td>
<td>f2</td>
<td>Half-precision floating point</td>
</tr>
<tr>
<td>float32</td>
<td>f4 or f</td>
<td>Standard single-precision floating point; compatible with C float</td>
</tr>
<tr>
<td>float64</td>
<td>f8 or d</td>
<td>Standard double-precision floating point; compatible with C double and Python float object</td>
</tr>
<tr>
<td>float128</td>
<td>f16 or g</td>
<td>Extended-precision floating point</td>
</tr>
<tr>
<td>complex64,</td>
<td>c8, c16,</td>
<td>Complex numbers represented by two 32, 64, or 128 floats, respectively</td>
</tr>
<tr>
<td>complex128,</td>
<td>c32</td>
<td></td>
</tr>
<tr>
<td>complex256</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bool</td>
<td>?</td>
<td>Boolean type storing True and False values</td>
</tr>
<tr>
<td>object</td>
<td>0</td>
<td>Python object type; a value can be any Python object</td>
</tr>
<tr>
<td>string_</td>
<td>S</td>
<td>Fixed-length ASCII string type (1 byte per character); for example, to create a string dtype with length 10, use 'S10'</td>
</tr>
<tr>
<td>unicode_</td>
<td>U</td>
<td>Fixed-length Unicode type (number of bytes platform specific); same specification semantics as string_ (e.g., 'U10')</td>
</tr>
</tbody>
</table>
**Speed Benefits**

- Compare random number generation in pure Python versus numpy
- Python:
  - import random
    
    ```
    %timeit rolls_list = [random.randrange(1,7) 
                         for i in range(0, 60_000)]
    ```
- With NumPy:
  - %timeit rolls_array = np.random.randint(1, 7, 60_000)
- Significant speedup (80x+)
Array Shape

• Our normal way of checking the size of a collection is... `len`

• How does this work for arrays?

• `arr1 = np.array([1,2,3,6,9])`
  `len(arr1) # 5`

• `arr2 = np.array([[1.5,2,3,4],[5,6,7,8]])`
  `len(arr2) # 2`

• All dimension lengths → shape: `arr2.shape # (2,4)`

• Number of dimensions: `arr2.ndim # 2`

• Can also reshape an array:
  - `arr2.reshape(4,2)`
  - `arr2.reshape(-1,2) # what happens here?`
Array Programming

• Lists:
  - c = []
    for i in range(len(a)):
      c.append(a[i] + b[i])

• How to improve this?
Array Programming

• Lists:
  - \( c = [] \)
    for \( i \) in \( \text{range}(\text{len}(a)) \):
      \( c.\text{append}(a[i] + b[i]) \)
  - \( c = [aa + bb \text{ for } aa, bb \text{ in } \text{zip}(a, b)] \)

• NumPy arrays:
  - \( c = a + b \)

• More functional-style than imperative
• **Internal iteration** instead of external
Operations

- $a = \text{np.array}([1, 2, 3])$
  $b = \text{np.array}([6, 4, 3])$

- (Array, Array) Operations (**Element-wise**)
  - Addition, Subtraction, Multiplication
  - $a + b \ # \ array([7, 6, 6])$

- (Scalar, Array) Operations (**Broadcasting**):
  - Addition, Subtraction, Multiplication, Division, Exponentiation
  - $a ** 2 \ # \ array([1, 4, 9])$
  - $b + 3 \ # \ array([9, 7, 6])$
More on Array Creation

- **Zeros:** `np.zeros(10)`
- **Ones:** `np.ones((4,5))` # shape
- **Empty:** `np.empty((2,2))`
- `_like versions:` pass an existing array and matches shape with specified contents
- **Range:** `np.arange(15)` # constructs an array, not iterator!
Indexing

• Same as with lists plus shorthand for 2D+
  - arr1 = np.array([6, 7, 8, 0, 1])
  - arr1[1]
  - arr1[-1]

• What about two dimensions?
  - arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])
  - arr[1][1]
  - arr[1,1] # shorthand
2D Indexing

In multidimensional arrays, if you omit later indices, the returned object will be a lower dimensional ndarray consisting of all the data along the higher dimensions. So in the $2 \times 2 \times 3$ array $\text{arr3d}$:

```
In [76]:
arr3d = np.array([[[1, 2, 3],
                   [4, 5, 6]],
                  [[7, 8, 9],
                   [10, 11, 12]]])
Out[76]:
array([[[ 1,  2,  3],
        [ 4,  5,  6]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

$\text{arr3d}[0]$ is a $2 \times 3$ array:

```
In [78]:
arr3d[0]
```

```
Out[78]:
array([[ 1,  2,  3],
        [ 4,  5,  6]])
```

Both scalar values and arrays can be assigned to $\text{arr3d}[0]$:

```
In [80]:
old_values = arr3d[0].copy()
In [81]:
arr3d[0] = 42
In [82]:
arr3d
```

```
Out[82]:
array([[[42, 42, 42],
        [42, 42, 42]],
       [[ 7,  8,  9],
        [10, 11, 12]]])
```

[W. McKinney, Python for Data Analysis]
Slicing

• 1D: Similar to lists
  - `arr1 = np.array([6, 7, 8, 0, 1])`
  - `arr1[2:5] # np.array([8,0,1]), sort of`

• Can **mutate** original array:
  - `arr1[2:5] = 3 # supports assignment`
  - `arr1 # the original array changed`

• Slicing returns **views** (copy the array if original array shouldn't change)
  - `arr1[2:5] # a view`
  - `arr1[2:5].copy() # a new array`
Slicing

• 2D+: comma separated indices as shorthand:
  - arr2 = np.array([[1.5, 2, 3, 4], [5, 6, 7, 8]])
  - a[1:3, 1:3]
  - a[1:3, :] # works like in single-dimensional lists

• Can combine index and slice in different dimensions
  - a[1, :] # gives a row
  - a[:, 1] # gives a column
Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as $==$) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048, 0.5433, -0.2349, 1.2792],
      [2.1452, 0.8799, -0.0523, 0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349, 1.2792]])
```

How to obtain the blue slice from array \texttt{arr}?
Figure 4-2. Two-dimensional array slicing

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as ==) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])
```

2D Array Slicing

How to obtain the blue slice from array arr?
2D Array Slicing

How to obtain the blue slice from array `arr`?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arr[2, 1:]</code></td>
<td>(2, 2)</td>
</tr>
<tr>
<td><code>arr[2]</code></td>
<td>(3,)</td>
</tr>
<tr>
<td><code>arr[2, :]</code></td>
<td>(3,)</td>
</tr>
<tr>
<td><code>arr[2:, :]</code></td>
<td>(1, 3)</td>
</tr>
</tbody>
</table>

[W. McKinney, Python for Data Analysis]
### 2D Array Slicing

#### Expression | Shape
--- | ---
arr[:2, 1:] | (2, 2)
arr[2] | (3,)
arr[2, :] | (3,)
arr[2:, :] | (1, 3)
arr[ :, 2] | (3, 2)

How to obtain the blue slice from array `arr`?

---

[W. McKinney, *Python for Data Analysis*]
2D Array Slicing

How to obtain the blue slice from array `arr`?

<table>
<thead>
<tr>
<th>Expression</th>
<th>Shape</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>arr[2, :]</code></td>
<td>(3,)</td>
</tr>
<tr>
<td><code>arr[2:]</code></td>
<td>(3,)</td>
</tr>
<tr>
<td><code>arr[2:, :]</code></td>
<td>(1, 3)</td>
</tr>
<tr>
<td><code>arr[:, 2]</code></td>
<td>(3, 2)</td>
</tr>
<tr>
<td><code>arr[1, :2]</code></td>
<td>(2,)</td>
</tr>
<tr>
<td><code>arr[1:2, :]</code></td>
<td>(1, 2)</td>
</tr>
<tr>
<td><code>arr[:2, 1:]</code></td>
<td>(2, 2)</td>
</tr>
</tbody>
</table>

Suppose each name corresponds to a row in the data array and we wanted to select all the rows with corresponding name 'Bob'. Like arithmetic operations, comparisons (such as `==`) with arrays are also vectorized. Thus, comparing names with the string 'Bob' yields a boolean array:

```
In [87]: names == 'Bob'
Out[87]: array([ True, False, False, True, False, False, False], dtype=bool)
```

This boolean array can be passed when indexing the array:

```
In [88]: data[names == 'Bob']
Out[88]:
array([[-0.048 ,  0.5433, -0.2349,  1.2792],
       [ 2.1452,  0.8799, -0.0523,  0.0672]])
```

The boolean array must be of the same length as the axis it's indexing. You can even mix and match boolean arrays with slices or integers (or sequences of integers, more on this later):

```
In [89]: data[names == 'Bob', 2:]
Out[89]:
array([[-0.2349,  1.2792]])
```

[W. McKinney, Python for Data Analysis]