Announcements - 01/13 (Thus)

• HW1-Part 1 posted on sakai: Resources -> Homworks -> HW1
  – Part 2 will have SQL queries and data analysis, and submission instructions
  – If you have not started working on it yet, start soon!
  – Both parts due on 01/27/2022 (Thursday)

• Threads for project teams posted on Ed
  – If you are looking for teammates or a team, please post
This is an overview and not exhaustive operations allowed in SQL

You will learn more as you run queries
Try on MovieLens data

Users-Ratings-Movies
Sailors-Reserved-Boats
Students-Enrolled-Courses

(entity-relationship-entity)
The SQL Query Language

• To find all 18 year old students, we can write:

```sql
SELECT * 
FROM Students S 
WHERE S.age = 18
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@ee</td>
<td>18</td>
<td>3.2</td>
</tr>
</tbody>
</table>

• To find just names and logins, replace the first line:

```sql
SELECT S.name, S.login
```

• To sort results, add at the end ORDER BY: ASC (default) or DESC

```sql
SELECT * 
FROM Students S 
WHERE S.age = 18 
ORDER BY gpa
```

ORDER BY gpa DESC, name
(first by gpa in descending order, then by name in ascending order)
Querying Multiple Relations

• What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade="A"
```

Given the following instances of Enrolled and Students:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

```
<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>
```

we get: ??
Querying Multiple Relations

• What does the following query compute?

```
SELECT S.name, E.cid
FROM Students S, Enrolled E
WHERE S.sid=E.sid AND E.grade="A"
```

Given the following instances of Enrolled and Students:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>

we get:

<table>
<thead>
<tr>
<th>S.name</th>
<th>E.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>Topology112</td>
</tr>
</tbody>
</table>
Basic SQL Query

```
SELECT      [DISTINCT]  <target-list>
FROM         <relation-list>
WHERE        <qualification>
```

- **relation-list**  A list of relation names
  - possibly with a “range variable” after each name
- **target-list**  A list of attributes of relations in relation-list
- **qualification**  Comparisons
  - (Attr op const) or (Attr1 op Attr2)
  - where op is one of = , <, >, <=, >= combined using AND, OR and NOT
- **DISTINCT**  is an optional keyword indicating that the answer should not contain duplicates
  - Default is that duplicates are not eliminated!
Conceptual Evaluation Strategy

- **Semantics** of an SQL query defined in terms of the following conceptual evaluation strategy:
  - Compute the cross-product of `<relation-list>`
  - Discard resulting tuples if they fail `<qualifications>`
  - Delete attributes that are not in `<target-list>`
  - If `DISTINCT` is specified, eliminate duplicate rows

- This strategy is probably the least efficient way to compute a query!
  - An optimizer will find more efficient strategies to compute the same answers
Example of Conceptual Evaluation

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

What does this query return?
Example of Conceptual Evaluation

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

Step 1: Form “cross product” of Sailor and Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

Reserves

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Example of Conceptual Evaluation

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```

Step 2: Discard tuples that do not satisfy <qualification>

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
**Step 3: Select the specified attribute(s)**

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

- **Sailors**
  - **sid** | **sname** | **rating** | **age**
  - 22 | dustin | 7 | 45
  - 31 | lubber | 8 | 55
  - 58 | rusty | 10 | 35

- **Reserves**
  - **sid** | **bid** | **day**
  - 22 | 101 | 10/10/96
  - 58 | 103 | 11/12/96

---

**Example of Conceptual Evaluation**

```sql
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid = 103
```
Recap

3. SELECT S.sname
1. FROM Sailors S, Reserves R
2. WHERE S.sid=R.sid AND R.bid=103

Always start from “FROM” -- form cross product
Apply “WHERE” -- filter out some tuples (rows)
Apply “SELECT” -- filter out some attributes (columns)

Ques. Does this get evaluated this way in practice in a Database Management System (DBMS)?

No! This is conceptual evaluation for finding what is correct!
We will learn about join and other operator algorithms later
A Note on “Range Variables”

- Sometimes used as a short-name
- The previous query can also be written as:

```sql
SELECT  S.sname
FROM    Sailors S, Reserves R
WHERE   S.sid=R.sid AND bid=103

OR

SELECT  sname
FROM    Sailors, Reserves
WHERE   Sailors.sid=Reserves.sid
        AND bid=103
```

It is good style, however, to use range variables always!
A Note on “Range Variables”

• Really needed only if the same relation appears twice in the FROM clause (called self-joins)

• Find pairs of Sailors of same age

```sql
SELECT S1.sname, S2.name
FROM Sailors S1, Sailors S2
WHERE S1.age = S2.age AND S1.sid < S2.sid
```

Why do we need the 2\textsuperscript{nd} condition?
Find sailor ids who’ve reserved at least one boat

```
SELECT ???
FROM   Sailors S, Reserves R
WHERE  S.sid=R.sid
```
Find sailor ids who’ve reserved at least one boat

SELECT S.sid  
FROM  Sailors S, Reserves R  
WHERE  S.sid=R.sid

• Would adding **DISTINCT** to this query make a difference?

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>
Find sailors who’ve reserved at least one boat

```sql
SELECT S.sid
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```

- Would adding `DISTINCT` to this query make a difference?
  - Note that if there are multiple bids for the same sid, you get multiple output tuples for the same sid
  - Without distinct, you get them multiple times

- What is the effect of replacing `S.sid` by `S.sname` in the `SELECT` clause?
  - Would adding `DISTINCT` to this variant of the query make a difference even if one sid reserves at most one bid?
Simple Aggregate Operators

Check yourself:
What do these queries compute?

SELECT COUNT (*)
FROM Sailors S

SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating=10

SELECT COUNT (DISTINCT S.rating)
FROM Sailors S
WHERE S.sname='Bob'

SELECT AVG (DISTINCT S.age)
FROM Sailors S
WHERE S.rating=10

SELECT S.sname
FROM Sailors S
WHERE S.rating= (SELECT MAX(S2.rating)
FROM Sailors S2)

COUNT (*)
COUNT ( [DISTINCT] A)
SUM ( [DISTINCT] A)
AVG ( [DISTINCT] A)
MAX (A)
MIN (A)

single column
CREATING / UPDATING TABLES
Creating Relations in SQL

- Creates the “Students” relation
  - the type (domain) of each field is specified
  - enforced by the DBMS whenever tuples are added or modified

- As another example, the “Enrolled” table holds information about courses that students take

CREATE TABLE Students
(sid CHAR(20),
name CHAR(20),
login CHAR(10),
age INTEGER,
gpa REAL)

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
gpa REAL)

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53831</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53831</td>
<td>Reggae203</td>
<td>B</td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
</tr>
</tbody>
</table>
Destroying and Altering Relations

DROP TABLE Students

• **Destroys the relation Students**
  
  – The schema information *and* the tuples are deleted.

ALTER TABLE Students
  
  ADD COLUMN firstYear: integer

• The schema of Students is altered by adding a new field; every tuple in the current instance is extended with a **NULL** value in the new field.
Adding and Deleting Tuples

• Can insert a single tuple using:

```
INSERT INTO Students (sid, name, login, age, gpa)
VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)
```

• Can delete all tuples satisfying some condition (e.g., name = Smith):

```
DELETE
FROM Students S
WHERE S.name = 'Smith'
```
Integrity Constraints (ICs)

- **IC**: condition that must be true for any instance of the database
  - e.g., domain constraints
  - ICs are specified when schema is defined
  - ICs are checked when relations are modified

- A legal instance of a relation is one that satisfies all specified ICs
  - DBMS will not allow illegal instances

- If the DBMS checks ICs, stored data is more faithful to real-world meaning
  - Avoids data entry errors, too!
Keys in a Database

• Key / Candidate Key
• Primary Key
• Super Key
• Foreign Key

• Primary key attributes are underlined in a schema
  – Person(pid, address, name)
  – Person2(address, name, age, job)
Primary Key Constraints

• A set of fields is a key for a relation if:
  1. No two distinct tuples can have same values in all key fields, and
  2. This is not true for any subset of the key

• Only Part 1 holds (Part 2 may be false)? A superkey
  – A key is also a superkey.

• If there are > 1 keys for a relation, one of the keys is chosen (by DBA = DB admin) to be the primary key
  – E.g., sid is a key for Students
  – The set \{sid, gpa\} is a superkey.

• Any possible benefit to refer to a tuple using primary key (than any key)?
Primary and Candidate Keys in SQL

• Possibly many candidate keys
  – specified using `UNIQUE`
  – one of which is chosen as the primary key.

• “For a given student and course, there is a single grade.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY ???)
Primary and Candidate Keys in SQL

• Possibly many candidate keys
  – specified using `UNIQUE`
  – one of which is chosen as the primary key.

• “For a given student and course, there is a single grade.”

```sql
CREATE TABLE Enrolled
    (sid CHAR(20),
     cid CHAR(20),
     grade CHAR(2),
     PRIMARY KEY (sid, cid)
    )
```
Primary and Candidate Keys in SQL

- Possibly many candidate keys
  - specified using `UNIQUE`
  - one of which is chosen as the primary key.

```
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid, cid)
)
```

```
CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARKEY ???, UNIQUE ???
)
```

- “For a given student and course, there is a single grade.”

- vs.

- “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”
Primary and Candidate Keys in SQL

- Possibly many candidate keys
  - specified using `UNIQUE`
  - one of which is chosen as the primary key.

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid))

CREATE TABLE Enrolled
(sid CHAR(20),
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY sid,
UNIQUE (cid, grade))
Primary and Candidate Keys in SQL

• Possibly many candidate keys
  – specified using `UNIQUE`
  – one of which is chosen as the primary key.

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid))

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY sid, UNIQUE (cid, grade))

• “For a given student and course, there is a single grade.”

• vs.

• “Students can take only one course, and receive a single grade for that course; further, no two students in a course receive the same grade.”

• Used carelessly, an IC can prevent the storage of database instances that arise in practice!
Foreign Keys, Referential Integrity

- **Foreign key**: Set of fields in one relation that is used to `refer` to a tuple in another relation
  - Must correspond to primary key of the second relation
  - Like a `logical pointer`

- **E.g. sid is a foreign key referring to Students:**
  - Enrolled(`sid`: string, `cid`: string, `grade`: string)
  - If all foreign key constraints are enforced, **referential integrity** is achieved
  - i.e., no dangling references
Foreign Keys in SQL

• Only students listed in the Students relation should be allowed to enroll for courses

CREATE TABLE Enrolled
(sid CHAR(20), cid CHAR(20), grade CHAR(2),
 PRIMARY KEY (sid,cid),
 FOREIGN KEY (sid) REFERENCES Students )

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>Reggae203</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>53650</td>
<td>Topology112</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>53666</td>
<td>History105</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Jones</td>
<td>jones@cs</td>
<td>18</td>
<td>3.4</td>
</tr>
<tr>
<td>53688</td>
<td>Smith</td>
<td>smith@eecs</td>
<td>18</td>
<td>3.2</td>
</tr>
<tr>
<td>53650</td>
<td>Smith</td>
<td>smith@math</td>
<td>19</td>
<td>3.8</td>
</tr>
</tbody>
</table>
Enforcing Referential Integrity

• Consider Students and Enrolled
  – sid in Enrolled is a foreign key that references Students.

• What should be done if an Enrolled tuple with a non-existent student id is inserted?
  – Reject it!

• What should be done if a Students tuple is deleted?
  – Three semantics allowed by SQL
    1. Also delete all Enrolled tuples that refer to it (cascade delete)
    2. Disallow deletion of a Students tuple that is referred to
    3. Set sid in Enrolled tuples that refer to it to a default sid
    4. (in addition in SQL): Set sid in Enrolled tuples that refer to it to a special value null, denoting `unknown’ or `inapplicable’

• Similar if primary key of Students tuple is updated
Referential Integrity in SQL

- SQL/92 and SQL:1999 support all 4 options on deletes and updates.
  - Default is **NO ACTION** (delete/update is rejected)
  - **CASCADE** (also delete all tuples that refer to deleted tuple)
  - **SET NULL / SET DEFAULT** (sets foreign key value of referencing tuple)

```sql
CREATE TABLE Enrolled
(sid CHAR(20) DEFAULT '000',
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY (sid,cid),
FOREIGN KEY (sid) REFERENCES Students
  ON DELETE CASCADE
  ON UPDATE SET DEFAULT )
```
Where do ICs Come From?

- ICs are based upon the semantics of the real-world enterprise that is being described in the database relations

- Can we infer ICs from an instance?
  - We can check a database instance to see if an IC is violated, but we can NEVER infer that an IC is true by looking at an instance.
  - An IC is a statement about all possible instances!
  - From example, we know name is not a key, but the assertion that sid is a key is given to us.

- Key and foreign key ICs are the most common; more general ICs supported too
Example Instances

• What does the key \((\text{sid, bid, day})\) in Reserves mean?

• If the key for the Reserves relation contained only the attributes \((\text{sid, bid})\), how would the semantics differ?

<table>
<thead>
<tr>
<th>Sailor</th>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Reserves</th>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
<td></td>
</tr>
</tbody>
</table>
• **TODOs:**

1. Start working on HW1-Part I:
   - Sakai -> Resources -> Homeworaks -> HW1

2. Read course policy ([link](#)) carefully before you start

3. Go to office hours if you have questions
   - Links on Ed

4. Check out the Project thread on Ed and keep looking for teams / teammates

5. Practice SQL on MovieLens, create / update new tables in a new database
Optional reading for SQL programming
Prepared statements: motivation

```python
while True:
    # Input bar, beer, price...
    cur.execute(""
    UPDATE Serves
    SET price = %s
    WHERE bar = %s AND beer = %s"
    (price, bar, beer))
    # Check result...
```

- Every time we send an SQL string to the DBMS, it must perform parsing, semantic analysis, optimization, compilation, and finally execution.
- A typical application issues many queries with a small number of patterns (with different parameter values).
- Can we reduce this overhead?
Prepared statements: example

```python
# Prepare once (in SQL).
PREPARE update_price AS
UPDATE Serves
SET price = $1
WHERE bar = $2 AND beer = $3"")
# parameter placeholders.
while True:
    # Input bar, beer, price...
    cur.execute('EXECUTE update_price(%s, %s, %s)',
                (price, bar, beer))
    # Note the switch back to %s for parameter placeholders.
    # Check result...
```

- The DBMS performs parsing, semantic analysis, optimization, and compilation only once, when it “prepares” the statement
- At execution time, the DBMS only needs to check parameter types and validate the compiled plan
- Most other API’s have better support for prepared statements than psycopg2
  - E.g., they would provide a `cur.prepare()` method
SQL Injection Attack

- The school probably had something like:
  ```python
  cur.execute("SELECT * FROM Students " + \
  "WHERE (name = " + name + ")")
  ```
  where `name` is a string input by user

- Suppose `name = Robert'; DROP TABLE Students; --`:
  - Drop deletes a table
  - `--` starts a comment
  - Becomes `SELECT * FROM Students WHERE (name = 'Robert'; DROP TABLE Students; -- ')`

[Image: xkcd.com/327/]

Duke CS, Spring 2022
CompSci 516: Database Systems
Guarding against SQL injection

- Escape certain characters in a user input string, to ensure that it remains a single string
  - E.g., ', which would terminate a string in SQL, must be replaced by '' (two single quotes in a row) within the input string
- Luckily, most API’s provide ways to “sanitize” input automatically (if you use them properly)
  - E.g., pass parameter values in psycopg2 through %s’s
- Check out Ashley Madison data breach story or https://medium.com/five-guys-facts/sql-injection-98199af86c9