This is an overview and not exhaustive operations allowed in SQL.

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

Announcements - 01/13 (Thus)

- HW1-Part 1 posted on sakai: Resources -> Homeworks -> 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

The SQL Query Language

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

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

  ```sql
  sid | name  | login    | age | gpa  
  --- |-------|----------|-----|------ 
  53666| Jones  | jones@cs | 18  | 3.4  
  53688| Smith  | smith@eecs | 18 | 3.2  
  53650| Smith  | smith@math | 19 | 3.8  
  ```

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

  ```sql
  SELECT S.name, S.login
  FROM Students S
  WHERE S.age = 18
  ```

  ```sql
  sid | name  | login    | age | gpa  
  --- |-------|----------|-----|------ 
  53666| Jones  | jones@cs | 18  | 3.4  
  53688| Smith  | smith@eecs | 18 | 3.2  
  53650| Smith  | smith@math | 19 | 3.8  
  ```

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

  ```sql
  SELECT S.name, S.login
  FROM Students S
  WHERE S.age = 18
  ```

  ```sql
  sid | name  | login    | age | gpa  
  --- |-------|----------|-----|------ 
  53666| Jones  | jones@cs | 18  | 3.4  
  53688| Smith  | smith@eecs | 18 | 3.2  
  53650| Smith  | smith@math | 19 | 3.8  
  ```

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

  ```sql
  SELECT S.name, S.login
  FROM Students S
  WHERE S.age = 18
  ```

  ```sql
  sid | name  | login    | age | gpa  
  --- |-------|----------|-----|------ 
  53666| Jones  | jones@cs | 18  | 3.4  
  53688| Smith  | smith@eecs | 18 | 3.2  
  53650| Smith  | smith@math | 19 | 3.8  
  ```

Querying Multiple Relations

- What does the following query compute?

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

  ```sql
  sid | name  | cid    | grade 
  --- |-------|-------|-------
  53831| Carnatic101 | C    
  53831| Reggae203  | B    
  53650| Topology112 | A    
  53666| History105  | B    
  ```

  Given the following instances of Enrolled and Students:

  ```sql
  sid | name  | login    | age | gpa  
  --- |-------|----------|-----|------ 
  53666| Jones  | jones@cs | 18  | 3.4  
  53688| Smith  | smith@eecs | 18 | 3.2  
  53650| Smith  | smith@math | 19 | 3.8  
  ```

  we get: ??

Querying Multiple Relations

- What does the following query compute?

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

  ```sql
  sid | name  | cid    | grade 
  --- |-------|-------|-------
  53831| Carnatic101 | C    
  53831| Reggae203  | B    
  53650| Topology112 | A    
  53666| History105  | B    
  ```

  Given the following instances of Enrolled and Students:

  ```sql
  sid | name  | login    | age | gpa  
  --- |-------|----------|-----|------ 
  53666| Jones  | jones@cs | 18  | 3.4  
  53688| Smith  | smith@eecs | 18 | 3.2  
  53650| Smith  | smith@math | 19 | 3.8  
  ```

  we get: ??
Basic SQL Query

\[
\text{SELECT} \ \ [\text{DISTINCT}] \ \ <\text{target-list}> \\
\text{FROM} \ \ <\text{relation-list}> \\
\text{WHERE} \ \ <\text{qualification}>
\]

- **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**
  - \((\text{Attr} \ op \ \text{const})\) or \((\text{Attr1} \ op \ \text{Attr2})\)
  - \(\text{where op is one of } , <, >, <=, >=\) combined using AND, OR and NOT
- **DISTINCT**: An optional keyword indicating that the answer should not contain duplicates
  - Default is that duplicates are not eliminated!

Example of Conceptual Evaluation

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

```
Sailor

<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>45</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55</td>
<td>8</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

What does this query return?
```

Example of Conceptual Evaluation

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

```
Sailor

<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>45</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55</td>
<td>8</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

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

```
                    | sid | sname | rating | age |
```
```
Sailor

<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>45</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55</td>
<td>8</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>
```

Example of Conceptual Evaluation

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

```
Sailor

<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>45</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55</td>
<td>8</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

Step 3: Select the specified attribute(s)
```

```
                    | sid | sname | rating | age |
```
```
Sailor

<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>45</td>
<td>7</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>55</td>
<td>8</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>
```

Conceptual Evaluation Strategy

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

- **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 \(<\text{qualifications}>\)
  - Delete attributes that are not in \(<\text{target-list}>\)
  - If \(<\text{qualification}>\) 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
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:

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

It is good style, however, to use range variables always!

Find sailor ids who’ve reserved at least one boat

• Would adding DISTINCT to this query make a difference?

Find sailors who’ve reserved at least one boat

• Would adding DISTINCT to this query make a difference?

• 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**

- `COUNT(*)`
- `COUNT(DISTINCT A)`
- `SUM(DISTINCT A)`
- `AVG(DISTINCT A)`
- `MAX(A)`
- `MIN(A)`

**Check yourself:**

What do these queries compute?

```
SELECT COUNT(*)
FROM Sailors S
```

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

```
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)
```

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

**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),
 grade CHAR(2))
```

**Destroying and Altering Relations**

- **DROP TABLE Students**
  - Destroys the relation Students.
  - The schema information and the tuples are deleted.

- **ALTER TABLE Students**
  - Adds a new field; every tuple in the current instance is extended with a NULL value in the new field.
  - 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.

**Integrity Constraints (ICs)**

- **IC:** condition that must be true for any instance of the database
  - E.g., domain constraints
  - ICs are specified when the 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

- **Part 2 false? 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.

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

- “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.”

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 ???)

- “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.”

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 ???)

- “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.”

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 ???)

- “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.”

CREATE TABLE Enrolled
  (sid CHAR(20),
   cid CHAR(20),
   grade CHAR(2),
   PRIMARY KEY (sid,cid, UNIQUE (sid, grade)))
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.”
- 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!

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

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

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

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, REFERENCES Students)
```

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

```
CREATE TABLE Enrolled
(sid CHAR(20) DEFAULT '000',
cid CHAR(20),
grade CHAR(2),
PRIMARY KEY sid, REFERENCES Students ON DELETE CASCADE ON UPDATE SET DEFAULT)
```

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)

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 (sid, bid, day) in Reserves mean?
- If the key for the Reserves relation contained only the attributes (sid, bid), how would the semantics differ?

**Sailor**

<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>

**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>

**Condition/Theta Join**

```
SELECT *
FROM Sailors S, Reserves R
WHERE S.sid=R.sid and age >= 40
```

Form cross product, discard rows that do not satisfy the condition

<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>31</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>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

**Next: different types of joins**

- Theta-join
- Equi-join
- Natural join
- Outer Join

**Equi Join**

```
SELECT *
FROM Sailors S, Reserves R
WHERE S.sid=R.bid and age = 45
```

A special case of theta join
Join condition only has equality predicate

<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>

**Outer Join**

```
SELECT S.sid, R.bid
FROM Sailors S LEFT OUTER JOIN Reserves R
ON S.sid=R.bid
```

Preserves all tuples from the left table whether or not there is a match
If no match, fill attributes from right with null
Similarly RIGHT/FULL outer join

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td></td>
</tr>
<tr>
<td>31</td>
<td>null</td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td></td>
</tr>
</tbody>
</table>
Expressions and Strings

```
SELECT S.age, age1=S.age-5, age2=S.age-5 AS age5
FROM Sailors S
WHERE S.sname LIKE 'B_5B'
```

- Illustrates use of arithmetic expressions and string pattern matching
- Find triples (of ages of sailors and two fields defined by expressions) for sailors
  - whose names begin and end with B and contain at least three characters
- `LIKE` is used for string matching, `'_'` stands for any one character and `'_%B'` stands for 0 or more arbitrary characters
  - You will need these often

Find sid's of sailors who've reserved a red and a green boat

```
Sailors (sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)
```

- UNION: Can be used to compute the union of any two union-compatible sets of tuples
  - can themselves be the result of SQL queries
- If we replace `OR` by AND in the first version, what do we get?
- Also available: `EXCEPT` (What do we get if we replace `UNION` by `EXCEPT`?)

Find sid's of sailors who've reserved a red or a green boat

```
Sailors (sid, sname, rating, age)
Reserves(sid, bid, day)
Boats(bid, bname, color)
```

- UNION: Can be used to compute the union of any two union-compatible sets of tuples
- `IN` is another set comparison operator, like IN
- Illustrates why, in general, subquery must be re-computed for each Sailor tuple

Find names of sailors who've reserved boat #103:

```
Sailors S
WHERE S.sid IN (SELECT R.sid
FROM Reserves R
WHERE R.bid=103)
```

- A very powerful feature of SQL:
  - a `WHERE/FROM/HAVING` clause can itself contain an SQL query
- To find sailors who've not reserved #103, use `NOT IN`.
- To understand semantics of nested queries, think of a nested loops evaluation
  - For each Sailors tuple, check the qualification by computing the subquery

Find names of sailors who've reserved boat #103:

```
Sailors S
WHERE EXISTS (SELECT R
FROM Reserves R
WHERE R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN
- Illustrates why, in general, subquery must be re-computed for each Sailor tuple

Nested Queries

```
Sailors S
WHERE EXISTS (SELECT R
FROM Reserves R
WHERE R.bid=103 AND S.sid=R.sid)
```

- Illustrates why, in general, subquery must be re-computed for each Sailor tuple

Nested Queries with Correlation

```
Sailors S
WHERE EXISTS (SELECT R
FROM Reserves R
WHERE R.bid=103 AND S.sid=R.sid)
```

- EXISTS is another set comparison operator, like IN
- Illustrates why, in general, subquery must be re-computed for each Sailor tuple
Recall: Aggregate Operators

- COUNT(*)
- SUM(DISTINCT A)
- AVG(DISTINCT A)
- MAX(A)
- MIN(A)

Group-by evaluation semantics

5. SELECT S.rating, MIN (S.age) AS minage
3. FROM Sailors S
2. WHERE S.age >= 18
3. GROUP BY S.rating
4. HAVING COUNT(*) > 1

More on Set-Comparison Operators

- We’ve already seen IN, EXISTS and UNIQUE
- Can also use NOT IN, NOT EXISTS and NOT UNIQUE.
- Also available: ANY, ALL, IN.

Group-By Evaluation semantics

- Expressions in HAVING must have a single value per group
- In effect, an attribute in HAVING is not an argument of an aggregate operator also appears in GROUP BY attributes list like, “GROUP BY bid, and HAVING bid = 2”.
- One answer tuple is generated per qualifying group
- A subset of GROUP BY attributes can appear in SELECT: SELECT A, SUM(C) FROM R GROUP BY A, B
Find age of the youngest sailor with age >= 18, for each rating with at least 2 such sailors.

Step 1: Form the cross product: FROM clause
(some attributes are omitted for simplicity)

Step 2: Apply WHERE clause

Step 3: Apply GROUP BY according to the listed attributes

Step 4: Apply HAVING clause

Step 5: Apply SELECT clause
Apply the aggregate operator
At the end, one tuple per group
Null Values

- Field values in a tuple are sometimes
  - unknown, e.g., a rating has not been assigned, or
  - inapplicable, e.g., no spouse’s name
- SQL provides a special value null for such situations.

Standard Boolean 2-valued logic

- True = 1, False = 0
- Suppose X = 5
  - (X < 100) AND (X >= 1) is $T \land T = T$
  - (X > 100) OR (X >= 1) is $F \lor T = T$
  - NOT(X = 5) is $\neg T = F$
- Intuitively,
  - $T = 1$, $F = 0$
  - For $V_1, V_2 \in \{1, 0\}$
  - $V_1 \land V_2 = \min(V_1, V_2)$
  - $V_1 \lor V_2 = \max(V_1, V_2)$
  - $\neg(V_1) = 1 \sim V_1$

2-valued logic does not work for nulls

- Suppose rating = null, X = 5
- Is rating > 8 true or false?
- What about AND, OR, and NOT connectives?
  - (rating > 8) AND (X = 5)?
- What if we have such a condition in the WHERE clause?

3-Valued Logic For Null

- TRUE (= 1), FALSE (= 0), UNKNOWN (= 0.5)
  - unknown is treated as 0.5
- Now you can apply rules from 2-valued logic!
  - For $V_1, V_2 \in \{1, 0, 0.5\}$
  - $V_1 \land V_2 = \min(V_1, V_2)$
  - $V_1 \lor V_2 = \max(V_1, V_2)$
  - $\neg(V_1) = 1 \sim V_1$
- Therefore,
  - NOT UNKNOWN = UNKNOWN
  - UNKNOWN OR TRUE = TRUE
  - UNKNOWN AND TRUE = UNKNOWN
  - UNKNOWN AND FALSE = FALSE
  - UNKNOWN OR FALSE = UNKNOWN

New issues for Null

- The presence of null complicates many issues. E.g.:
  - Special operators needed to check if value IS/IS NOT NULL
  - Be careful!
  - “WHERE X = NULL” does not work!
  - Need to write “WHERE X IS NULL”
- Meaning of constructs must be defined carefully
  - e.g., WHERE clause eliminates rows that don’t evaluate to true
  - So not only FALSE, but UNKNOWNS are eliminated too
  - very important to remember!
- Arithmetic with NULL
  - all of +, -, *, / return null if any argument is null
- Can force “no nulls” while creating a table
  - name char(20) NOT NULL
  - primary key is always not null

Aggregates with NULL

- What do you get for
  - SELECT count(*) from R1?
  - SELECT count(rating) from R1?

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

R1
Aggregates with NULL

- What do you get for
  - SELECT count(*) from R1?
  - SELECT count(rating) from R1?

  Ans: 3 for both

COUNT, SUM, AVG, MIN, MAX (with or without DISTINCT)
- Discards null values first
- Then applies the aggregate
- Except count(*)

- If only applied to null values, the result is null

Can create a new table from a query on other tables too

```
SELECT * INTO ... FROM ... WHERE
```

A view is just a relation, but we store a definition, rather than a set of tuples

```
CREATE VIEW YoungActiveStudents (name, grade)
AS
  SELECT S.name, E.grade
  FROM Students S, Enrolled E
  WHERE S.sid = E.sid AND S.age < 21
```

- Views can be dropped using the DROP VIEW command
- Views and Security: Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s)
  - the above view hides courses "cid" from E
- More on views later in the course
“WITH” clause – very useful!

- You will find “WITH” clause very useful!

```
WITH Temp1 AS
(SELECT ..... .),
Temp2 AS
(SELECT ..... .)
SELECT X, Y
FROM Temp1, Temp2
WHERE ....
```

- Can simplify complex nested queries

Overview: General Constraints

- Useful when more general ICs than keys are involved

```
CREATE TABLE Sailors
(sid INTEGER,
name: CHAR(10),
rating INTEGER,
age: REAL,
PRIMARY KEY (sid),
CHECK (rating >= 1 AND rating <= 10)
```

- There are ASSERTIONS to specify constraints that span across multiple tables

```
CREATE TABLE Reserves
(sname: CHAR(10),
bid: INTEGER,
day: DATE,
PRIMARY KEY (bid, day),
CONSTRAINT noInterlakeRes
CHECK (`Interlake' <> (SELECT B.name FROM Boats B
WHERE B.bid=bid))
```

- There are TRIGGERS too: procedure that starts automatically if specified changes occur to the DBMS

```
CREATE TRIGGER youngSailorUpdate
AFTER INSERT ON SAILORS
REFERENCING NEW TABLE NewSailors
FOR EACH STATEMENT
INSERT INTO YoungSailors (sid, name, age, rating)
SELECT sid, name, age, rating
FROM NewSailors
WHERE age <= 18
```

Summary: SQL

- SQL has a huge number of constructs and possibilities
  - You need to learn and practice it on your own
  - Given a problem, you should be able to write a SQL query and verify whether a given one is correct

- Pay attention to NULLs

- Can limit answers using “LIMIT” or “TOP” clauses
  - e.g. to output TOP 20 results according to an aggregate
  - also can sort using ASC or DESC keywords

Summary

- Relational Data

- SQL
  - Semantic
  - Join
  - Simple Aggregates
  - Nested Queries

- You will learn these further and run yourself on PostGres on Thursday in the in-class lab on SQL!

End of Lecture-3 (01/13)

- TODOs:
  1. Start working on HW1-Part I:
     - Sakai -> Resources -> Homeworks -> 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
Optional reading for SQL programming

Prepared statements: motivation

- 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

- 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 APIs have better support for prepared statements than psycopg2
  - E.g., they would provide a `cur.prepares()` method

SQL Injection Attack

- The school probably had something like:
  ```sql
  cur.execute("SELECT * FROM Students " + 
  "WHERE (name = " + name + "); DROP TABLE Students; --")
  ```
  where `name` is a string input by user
- Suppose `name = Robert'; DROP TABLE Students;
  --` starts a comment
- Becomes `SELECT * FROM Students WHERE (name = 'Robert'; DROP TABLE Students; --')`

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