CompSci 516
Database Systems

Lecture 4
SQL
and
Relational Calculus

Instructor: Sudeepa Roy
Announcements

• Lab-1 makeup instructions sent on piazza
  – Please respond by 3 pm today if you have missed the lab

• Let me know if you are still not on piazza

• HW1 will be posted after the class
  – On sakai (data is already there)
  – Deadlines in stages
  – First deadline on 09/17
Today’s topic

• Finish SQL
• RC

• Next week:
  – Tuesday: Guest Lecture by Junyang Gao: RA
  – Thursday: Lab on RA

Acknowledgement:
The following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.
Nulls and Views in SQL
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**
  - \((X < 100) \text{ AND } (X \geq 1)\) is \(T \land T = T\)
  - \((X > 100) \text{ OR } (X \geq 1)\) is \(F \lor T = T\)
  - \((X > 100) \text{ AND } (X \geq 1)\) is \(F \land T = F\)
  - NOT\((X = 5)\) is \(\neg T = F\)

- **Intuitively,**
  - \(T = 1, F = 0\)
  - For \(V1, V2 \in \{1, 0\}\)
    - \(V1 \land V2 = \text{MIN}(V1, V2)\)
    - \(V1 \lor V2 = \text{MAX}(V1, V2)\)
    - \(\neg(V1) = 1 - V1\)
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 V1, V2 ∈ \{1, 0, 0.5\}
  - V1 ∧ V2 = \text{MIN}(V1, V2)
  - V1 ∨ V2 = \text{MAX}(V1, V2)
  - ¬(V1) = 1 − V1

- 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!

• But NULL allows new operators (e.g. outer joins)

• Can force ”no nulls” while creating a table
  – sname 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`?

<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>
Aggregates with NULL

<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>

R1

- What do you get for
- SELECT count(*) from R1?
- SELECT count(rating) from R1?
- Ans: 3 for both
Aggregates with NULL

What do you get for
- SELECT count(*) from R1?
- SELECT count(rating) from R1?
- Ans: 3 for both

What do you get for
- SELECT count(*) from R2?
- SELECT count(rating) from R2?
## Aggregates with NULL

### R1

<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>

- What do you get for
- \( \text{SELECT count(*) from R1?} \)
- \( \text{SELECT count(rating) from R1?} \)
- **Ans:** 3 for both

### R2

<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>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

- What do you get for
- \( \text{SELECT count(*) from R2?} \)
- \( \text{SELECT count(rating) from R2?} \)
- **Ans:** First 3, then 2
Aggregates with NULL

- **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

### R2

<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>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35</td>
</tr>
</tbody>
</table>

**SELECT sum(rating) from R2?**

- **Ans: 17**

### R3

<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>null</td>
<td>45</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>null</td>
<td>55</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>null</td>
<td>35</td>
</tr>
</tbody>
</table>

**SELECT sum(rating) from R3?**

- **Ans: null**
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
  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:

  \[
  \text{INSERT INTO Students (sid, name, login, age, gpa)} \\
  \text{VALUES (53688, 'Smith', 'smith@ee', 18, 3.2)}
  \]

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

  \[
  \text{DELETE} \\
  \text{FROM Students S} \\
  \text{WHERE S.name = 'Smith'}
  \]
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

• 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.

• “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.”

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),
PRIMARY KEY ???,
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))

• “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.

• “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:**
  – 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>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>Carnatic101</td>
<td>C</td>
</tr>
<tr>
<td>53666</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>

<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.
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?

<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>
Views

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

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

```
SELECT S.name, E.grade
INTO YoungActiveStudents
FROM Students S, Enrolled E
WHERE S.sid = E.sid and S.age<21
```
“WITH” clause – very useful!

- You will find “WITH” clause very useful!

```sql
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

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

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

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

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

• SQL has a huge number of constructs and possibilities
  – You need to learn and practice it on your own

• 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

• We learnt
  – Creating/modifying relations
  – Specifying integrity constraints
  – Key/candidate key, superkey, primary key, foreign key
  – Conceptual evaluation of SQL queries
  – Joins
  – Group bys and aggregates
  – Nested queries
  – NULLs
  – Views
Relational Query Languages
Relational Query Languages

• Query languages: Allow manipulation and retrieval of data from a database

• Relational model supports simple, powerful QLs:
  – Strong formal foundation based on logic
  – Allows for much optimization

• Query Languages != programming languages
  – QLs not intended to be used for complex calculations
  – QLs support easy, efficient access to large data sets
Formal Relational Query Languages

• Two “mathematical” Query Languages form the basis for “real” languages (e.g. SQL), and for implementation:
  – Relational Calculus: Lets users describe what they want, rather than how to compute it (Non-operational, declarative, or procedural)
  – Relational Algebra: More operational, very useful for representing execution plans

• Note: Declarative (RC, SQL) vs. Operational (RA)
Relational Calculus (RC)
Logic Notations

• $\exists$ There exists
• $\forall$ For all
• $\land$ Logical AND
• $\lor$ Logical OR
• $\neg$ NOT
• $\Rightarrow$ Implies
TRC: example

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

• Find the name and age of all sailors with a rating above 7

\{P \mid \exists \ S \in \text{Sailors} \ (\text{S.rating} > 7 \land P.\text{sname} = S.\text{sname} \land P.\text{age} = S.\text{age})\}\}

• P is a tuple variable
  – with exactly two fields sname and age (schema of the output relation)
  – P.sname = S.sname \land P.age = S.age gives values to the fields of an answer tuple

• Use parentheses, \forall \ \exists \ \lor \ \land \ > \ < \ = \ \neq \ \neg \ etc \ as \ necessary

• A \Rightarrow B \ is \ very \ useful \ too
  – next slide
A \Rightarrow B

- A “implies” B
- Equivalently, if A is true, B must be true
- Equivalently, \neg A \lor B, i.e.
  - either A is false (then B can be anything)
  - otherwise (i.e. A is true) B must be true
Useful Logical Equivalences

• $\forall x \ P(x) = \neg \exists x \ [\neg P(x)]$

• $\neg (P \lor Q) = \neg P \land \neg Q$
• $\neg (P \land Q) = \neg P \lor \neg Q$
  - Similarly, $\neg (\neg P \lor Q) = P \land \neg Q$ etc.

• $A \Rightarrow B = \neg A \lor B$
TRC: example

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

• Find the names of sailors who have reserved at least two boats
TRC: example

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

• Find the names of sailors who have reserved at least two boats

\{P \mid \exists S \in \text{Sailors} \ (\exists R1 \in \text{Reserves} \ \exists R2 \in \text{Reserves} \ (S.sid = R1.sid \ \land \ S.sid = R2.sid \ \land \ R1.bid \neq R2.bid) \ \land \ P.sname = S.sname)\}
TRC: example

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

- Find the names of sailors who have reserved all boats
- Called the “Division” operation


TRC: example

Sailors\((\text{sid}, \text{sname}, \text{rating}, \text{age})\)
Boats\((\text{bid}, \text{bname}, \text{color})\)
Reserves\((\text{sid}, \text{bid}, \text{day})\)

- Find the names of sailors who have reserved all boats
- Division operation in RA!

\{\text{P} \mid \exists S \in \text{Sailors} [\forall B \in \text{Boats} (\exists R \in \text{Reserves} (S.\text{sid} = R.\text{sid} \land R.\text{bid} = B.\text{bid}))] \land (\text{P.sname} = S.\text{sname})}
TRC: example

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

- Find the names of sailors who have reserved all red boats

How will you change the previous TRC expression?
TRC: example

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

- Find the names of sailors who have reserved all red boats

\{P \mid \exists S \in \text{Sailors} \ (\forall B \in \text{Boats} \ (B.\text{color} = \text{'red'} \Rightarrow (\exists R \in \text{Reserves} \\
S.\text{sid} = R.\text{sid} \land R.\text{bid} = B.\text{bid}))) \land P.\text{name} = S.\text{name})\}

Recall that $A \Rightarrow B$ is logically equivalent to $\neg A \lor B$
so $\Rightarrow$ can be avoided, but it is cleaner and more intuitive
More Examples: RC

• The famous “Drinker-Beer-Bar” example!

UNDERSTAND THE DIFFERENCE IN ANSWERS FOR ALL FOUR DRINKERS

Drinker Category 1

Find drinkers that frequent some bar that serves some beer they like.
Drinker Category 1

Find drinkers that frequent some bar that serves some beer they like.

\[
Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z)
\]

a shortcut for
\[
\{x \mid \exists Y \in \text{Frequents} \exists Z \in \text{Serves} \exists W \in \text{Likes} ((T.\text{drinker} = x.\text{drinker}) \land (T.\text{bar} = Z.\text{bar}) \land (W.\text{beer} = Z.\text{beer}) \land (Y.\text{drinker} = W.\text{drinker}) \}
\]

The difference is that in the first one, one variable = one attribute
in the second one, one variable = one tuple (Tuple RC)
Both are equivalent and feel free to use the one that is convenient to you
Drinker Category 2

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent only bars that serves some beer they like.

\[ Q(x) = \ldots \]
Drinker Category 2

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent only bars that serves some beer they like.

\[ Q(x) = \forall y. \text{Frequents}(x, y) \Rightarrow (\exists z. \text{Serves}(y, z) \land \text{Likes}(x, z)) \]
Drinker Category 3

Find drinkers that frequent **some** bar that serves **some** beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent **only** bars that serves **some** beer they like.

\[ Q(x) = \forall y. \text{Frequents}(x, y) \Rightarrow (\exists z. \text{Serves}(y, z) \land \text{Likes}(x, z)) \]

Find drinkers that frequent **some** bar that serves **only** beers they like.

\[ Q(x) = \ldots \]
Find drinkers that frequent some bar that serves some beer they like.

$$Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z)$$

Find drinkers that frequent only bars that serves some beer they like.

$$Q(x) = \forall y. \text{Frequents}(x, y) \Rightarrow (\exists z. \text{Serves}(y, z) \land \text{Likes}(x, z))$$

Find drinkers that frequent some bar that serves only beers they like.

$$Q(x) = \exists y. \text{Frequents}(x, y) \land \forall z. (\text{Serves}(y, z) \Rightarrow \text{Likes}(x, z))$$
Drinker Category 4

Find drinkers that frequent **some** bar that serves **some** beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent **only** bars that serve **some** beer they like.

\[ Q(x) = \forall y. \text{Frequents}(x, y) \Rightarrow (\exists z. \text{Serves}(y, z) \land \text{Likes}(x, z)) \]

Find drinkers that frequent **some** bar that serves **only** beers they like.

\[ Q(x) = \exists y. \text{Frequents}(x, y) \land \forall z. (\text{Serves}(y, z) \Rightarrow \text{Likes}(x, z)) \]

Find drinkers that frequent **only** bars that serve **only** beer they like.

\[ Q(x) = \ldots \]
Drinker Category 4

Find drinkers that frequent some bar that serves some beer they like.

\[ Q(x) = \exists y. \exists z. \text{Frequents}(x, y) \land \text{Serves}(y, z) \land \text{Likes}(x, z) \]

Find drinkers that frequent only bars that serves some beer they like.

\[ Q(x) = \forall y. \text{Frequents}(x, y) \Rightarrow (\exists z. \text{Serves}(y, z) \land \text{Likes}(x, z)) \]

Find drinkers that frequent some bar that serves only beers they like.

\[ Q(x) = \exists y. \text{Frequents}(x, y) \land \forall z. (\text{Serves}(y, z) \Rightarrow \text{Likes}(x, z)) \]

Find drinkers that frequent only bars that serves only beer they like.

\[ Q(x) = \forall y. \text{Frequents}(x, y) \Rightarrow \forall z. (\text{Serves}(y, z) \Rightarrow \text{Likes}(x, z)) \]
Why should we care about RC

- RC is declarative, like SQL, and unlike RA (which is operational)
- Gives foundation of database queries in first-order logic
  - you cannot express all aggregates in RC, e.g. cardinality of a relation or sum (possible in extended RA and SQL)
  - still can express conditions like “at least two tuples” (or any constant)
- RC expression may be much simpler than SQL queries
  - and easier to check for correctness than SQL
  - power to use \( \forall \) and \( \Rightarrow \)
  - then you can systematically go to a “correct” SQL query
Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

From RC to SQL

Query: Find drinkers that like some beer (so much) that they frequent all bars that serve it

$$Q(x) = \exists y. \text{Likes}(x, y) \land \forall z. (\text{Serves}(z, y) \Rightarrow \text{Frequents}(x, z))$$
From RC to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

\[ Q(x) = \exists y. \text{Likes}(x, y) \land \forall z. (\text{Serves}(z, y) \Rightarrow \text{Frequents}(x, z)) \]

\[ \equiv Q(x) = \exists y. \text{Likes}(x, y) \land \forall z. (\neg \text{Serves}(z, y) \lor \text{Frequents}(x, z)) \]

Step 1: Replace \( \forall \) with \( \exists \) using de Morgan’s Laws

\[ Q(x) = \exists y. \text{Likes}(x, y) \land \neg \exists z. (\text{Serves}(z, y) \land \neg \text{Frequents}(x, z)) \]
From RC to SQL

Query: Find drinkers that like some beer so much that they frequent all bars that serve it

\[ Q(x) = \exists y. \text{Likes}(x, y) \land \neg \exists z. (\text{Serves}(z, y) \land \neg\text{Frequents}(x, z)) \]

Step 2: Translate into SQL

```sql
SELECT DISTINCT L.drinker
FROM Likes L
WHERE not exists
  (SELECT S.bar
   FROM Serves S
   WHERE L.beer=S.beer
   AND not exists (SELECT *
                   FROM Frequents F
                   WHERE F.drinker=L.drinker
                   AND F.bar=S.bar))
```

We will see a “methodical and correct” translation through “safe queries” in Datalog