NoSQL, Security, Miscellaneous, and General Review

Oct. 14

Logistics

- Good job finishing P2 and HW3
- Read some documentations on MongoDB if you want to get a head-start on P3

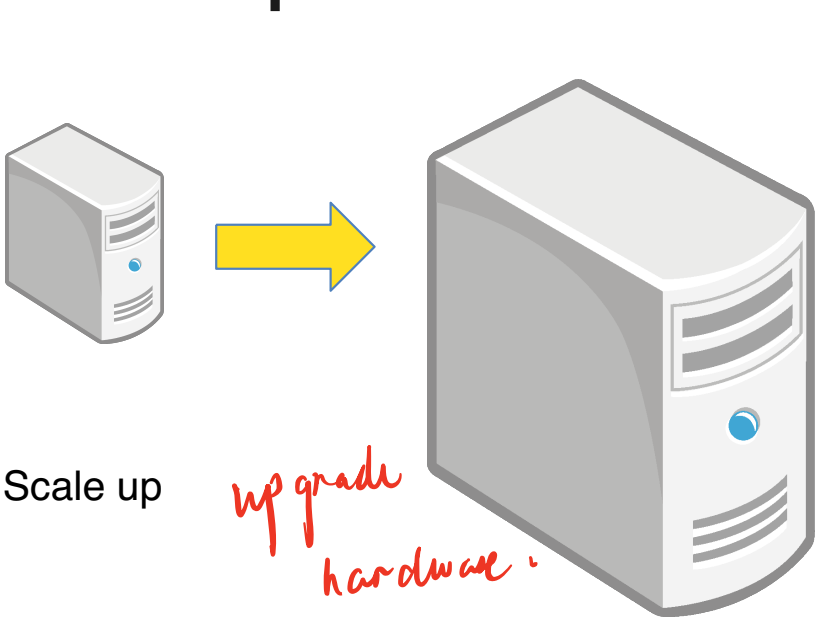
- Midterm on Oct. 21, 7:30 - 9:30 PM EST
  - Mix of true/false, multiple choice, fill in blanks, live-coding
  - One double sided 8.5x11 **handwritten** cheat-sheet is allowed
- Review session Oct. 19, 3:00 - 6:00 PM EST on Zoom, recorded
- Practice exams are available on Canvas
- We will go over conceptual material and example problems today
- No past exam covered today (will be covered on Oct. 19)

Study Resources

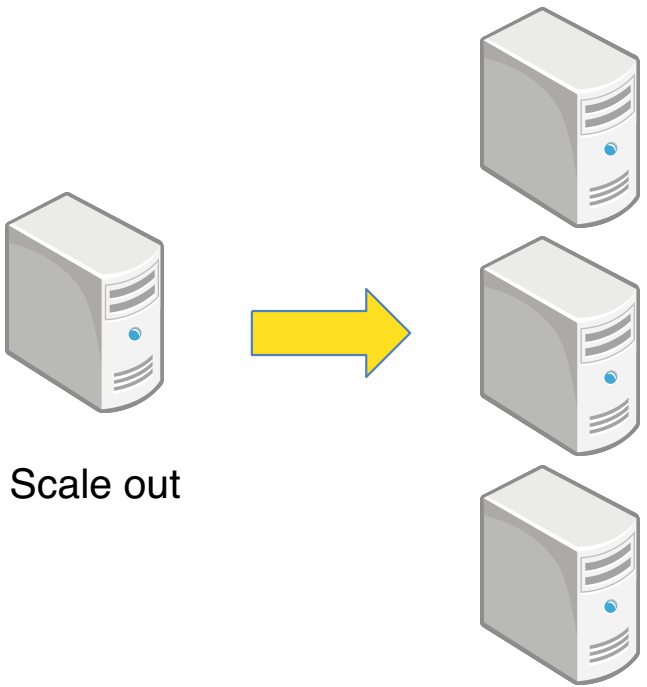
- Lectures 1 - 11
- Discussions 1 - 6
- Projects 1 and 2
- Homework 1 - 3
- This review session + GSI review session
- Past exams on Canvas
- Piazza
- Textbook
- Office hours

- ER Diagrams
  - Draw and interpret
- SQL
  - Create / insert / drop tables
  - Performing queries
- Relational Algebra / Calculus
  - Write & interpret RA, interpret RC
- Normalization
  - 3NF, BCNF decomposition
  - Candidate keys
  - Lossless join, dependency preserving
- Miscellaneous
  - NoSQL, dbApps, security, privacy, responsibility
  - Best practices

Scale-up vs Scale-out



Scale up



Scale out

Scale-out

- Partitioning
  - Distribute data across many servers (low redundancy)
  - **Hopefully
    - Evenly distribute across servers
    - Minimize trax / data transfer across servers (slow)
  - May not exist a good partitioning
- Replication
  - Data is replicated across multiple servers
  - High performance, high availability, high redundancy
  - Synchronous replication
    - Acknowledges client after ALL replicas updated successfully
    - 1CS -> but bad for performance, lack of fault tolerance, and bottleneck
  - Asynchronous replication
    - Acknowledges first, then update
    - Not 1CS -> inconsistencies, less guarantees
- 1-Copy Serializability (1CS)
  - System behaves like a serial processor of transactions on a single copy database

sharding is not guarantee to ↑ performance

⇒ don't have to communicate bw servers as much as before

NoSQL

- Does not mean that we do not use SQL language
- Rather means " not only SQL"
- Data-intensive systems that store data other than the traditional relational tables
  - Key-value stores
  - Document stores
  - SQL-on-MapReduce
  - Graph databases
  - Columnar databases
- Many pros and cons vs traditional relational databases (many are double-edged swords)
- Relational databases still the most popular
- MapReduce: filters a large sum of data, then organize and reduce them
  - If you're interested, check out lecture 11 from EECS 485

NoSQL continued...

- Pros
  - Flexible schema
  - Easy setup
  - Scalability
  - Higher performance and availability (trade-off with consistency)
  - Lower licensing costs
  - Less complicated logics such as joins, set operations, etc. (lack of support can also be a con)
- Cons
  - Less structured schema
  - No declarative language (SQL) (usually) -> Lower level of abstraction -> More coding
  - Less consistency -> Fewer guarantees -> harder coding
  - Lack of standards (many different APIs and platforms)
  - No support for joins (could also be a pro in some cases)
- ***Important: these are usually true. NoSQL varies a lot, and there can be exceptions

OLTP vs OLAP

- Online Transactional Processing (OLTP): day-to-day operations
- Online Analytical Processing (OLAP): aggregations, statistics, slicing and dicing

	OLTP	OLAP
Latency	< 1 second	mins-days
Read-Writes	Small reads + writes	Read-Only (bulk loads)
Initiated By:	(Usually) end-user activities	Analysts / Reporting Tools
Predictability	Instances of a few templates	Ad-hoc (unless issued by reporting tools)
Degree of concurrency	100's-100K's high	10's (<100)
Potential Bottlenecks	I/O, locks, CPU, network	(network or disk) I/O
Size	Moderate (Working set can fit in memory)	No limit (can be in PB's)

Database Security

- Database security
  - SQL injections
    - Maliciously input code instead of intended parameters to query functions
  - Weak password storage
    - Poor encryption, lack of salting, etc.
  - Too much privileges to users
    - Imagine if many Google employees have full access of the entire Google database
- Security best practices
  - Sanitize query inputs
    - How do you distinguish between code vs valid user input?
  - Use established and robust encryption methods (take EECS 485, EECS 475, EECS 388 to learn more)
  - Principle of least privilege
    - Give users only privileges they absolutely need
- https://www.darkreading.com/vulnerabilities-threats/the-10-most-common-database-vulnerabilities

SQL Injection

- Users can inject SQL code in place of an intended user input
- Example: the below Python program is vulnerable (no need to know Python)

```
# Intends to recover one's password using a one-time recovery code
def make_query(recovery_code): # recovery_code is a string parameter
    stmt = 'SELECT password FROM Users_info '
           f'WHERE code = {recovery_code}'
    return stmt
```

- What if recovery_code == " 10 OR 1 = 1"
  - Steal passwords without knowing the recovery code
- What if recovery_code == " 10; DROP TABLE Users_info"
  - Delete all data
- https://www.w3schools.com/sql/sql_injection.asp

Data Responsibility and Privacy

- Your data is everywhere!
  - Impossible to leave no internet footsteps
- Many ways to get close to anonymity (still not close enough sometimes)
  - K-Anonymity
    - Require at least k entries in a group about which information is revealed
    - Ensure tuple cannot be distinguished from k-1 other tuples
  - Differential Privacy
    - Rigorous mathematical and algorithmic
  - Learn more from the end of EECS 485 lecture 6
- Algorithms can be biased
- Data equality vs. equity (both have their purposes)
  - Equality: training dataset is an unbiased sample of the population: groups are represented in proportion to their sizes in the entire population
  - Equity: may require over-sampling of small minorities ⇒ minimize dominant effect of majority

Functional Dependencies and Redundancy

- Less tables != less redundant
  - Example: p1 public dataset. Only 1 table for public_user_information, but many redundancies
- Want: break large datasets into smaller and less redundant tables
- 3NF and BCNF ensure lossless join (required)
- Dependency-preserving is not guaranteed (nice to have)
  - Always possible for 3NF
  - Not always for BCNF
- Clarification on redundancy: @172 on Piazza

Functional Dependency Example

- Take a look at the cities table we designed in P1
  - Columns are city_id, name, state, country
  - City_id is the primary key
    - city_id ⇒ (city_id, name, state, country, avg_temp, time_zone, latitude, longitude)
  - Each combination of (name, state, country) is also unique
    - (name, state, country) ⇒ (city_id, name, state, country, avg_temp, time_zone, latitude, longitude)
  - Let's add columns avg_temp, time_zone, latitude, longitude
    - Given a longitude, you know a city's time zone (1)
    - Given a latitude, you can figure out the avg_temp with certainty (2)
    - Given the combination of time zone and avg_temp, you can find out a corresponding city_id (3)
  - List functional dependencies you can find from the above 3 statements
    - Longitude ⇒ time_zone (1)
    - Latitude ⇒ avg_temp (2)
    - (Time_zone, avg_temp) ⇒ city_id (3)

Functional Dependency Solution

- Find all candidate keys from the functional dependencies
  - City_id, (name, state, country) both functionally determine ALL attributes
    - City_id is minimal, check if a subset of (name, state, country) is a candidate key
    - Turns out no subset is a candidate key, which means (name, state, country) is minimal
  - (latitude, longitude) functionally determines ALL attributes
    - Latitude ⇒ avg_temp
    - Longitude ⇒ time_zone
    - Which means that (latitude, longitude) ⇒ (avg_temp, time_zone)
    - (latitude, longitude) ⇒ (avg_temp, time_zone) ⇒ city_id ⇒ (all columns)
    - (avg_temp, time_zone) ⇒ **all attributes**
    - (latitude, longitude) ⇒ **all attributes**
    - Check if either latitude, longitude, avg_temp, time_zone are candidate keys by themselves
    - Answer is no for all, so (latitude, longitude) and (avg_temp, time_zone) are candidate keys
  - Final answer: city_id, (latitude, longitude), (avg_temp, time_zone), (name, state, country)

Functional Dependency Example 3NF? BCNF?

- Longitude ⇒ time_zone (1)
- Latitude ⇒ avg_temp (2)
- (Time_zone, avg_temp) ⇒ city_id (3)
- City_id ⇒ **all attributes** (4)
- (name, state, country) ⇒ **all attributes** (5)
- Candidate keys: city_id, (latitude, longitude), (avg_temp, time_zone), (name, state, country)

Relation R with FDs F is in 3NF if, for all $X \rightarrow A$ in F^+

- $A \subseteq X$ (trivial dependency) or
- X is a super key or
- A is part of some (minimal) key for R **(prime attribute)**

X: subset of attributes

A: single attribute

Rel. R with FDs F is in BCNF if, for all $X \rightarrow A$ in F^+

- $A \subseteq X$ (trivial FD), or
- X is a super key

X: subset of attributes

A: single attribute

Functional Dependency Example 3NF? BCNF? Solution

- 3NF
  - For (1), (2), time_zone, avg_temp are both part of a minimal key (avg_temp, time_zone)
  - For (3), (time_zone, avg_temp) is a candidate key, thus super key
  - For (4), (5), we can break the **all attributes** up to many smaller relations in the form of city_id ⇒ attribute1, city_id ⇒ attribute2, city_id ⇒ attribute3, etc.
  - Every one of those smaller relations are valid because city_id and (name, state, country) are super keys
  - Relation is 3NF
- BCNF
  - For (1), (2), time_zone and avg_temp are not subsets of longitude and latitude respectively
  - Longitude and latitude are not super keys by themselves
  - Rest of the dependencies are valid
  - Relation is not BCNF
  - Perform a BCNF decomposition! Post on Piazza if you have any questions

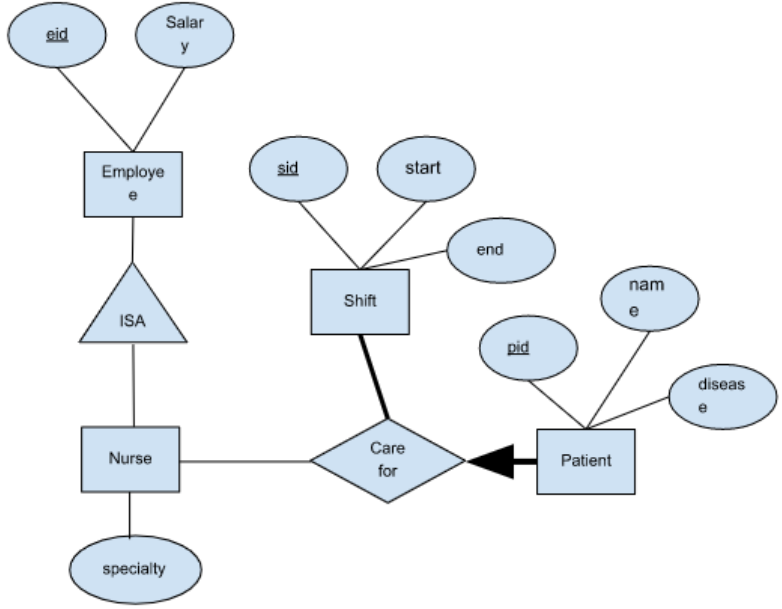
ER Diagrams

- @66 on Piazza
- For binary relationships
  - ***Each <entity A> can *participate in relationships* with <any number / at least 1 / at most 1 / exactly 1 (according to the line that connects to entity A)> of <entity B>***
- For ternary relationships / relationships involving >3 entities
  - ***Each <entity A> can *participate in relationships* with <any number / at least 1 / at most 1 / exactly 1 (according to the line that connects to entity A)> of <entity B> and <entity C> and ……***
- Put those on your cheat-sheet if you're struggling to memorize how to interpret complex relationships
- Make sure you look at 1 relationship at a time
  - For example, Q4e on HW1
- Attributes of relationships CANNOT be part of a key for the relationship
  - For example, Q4b
  - Half the students got it wrong

ER Diagrams: Ternary Relationships

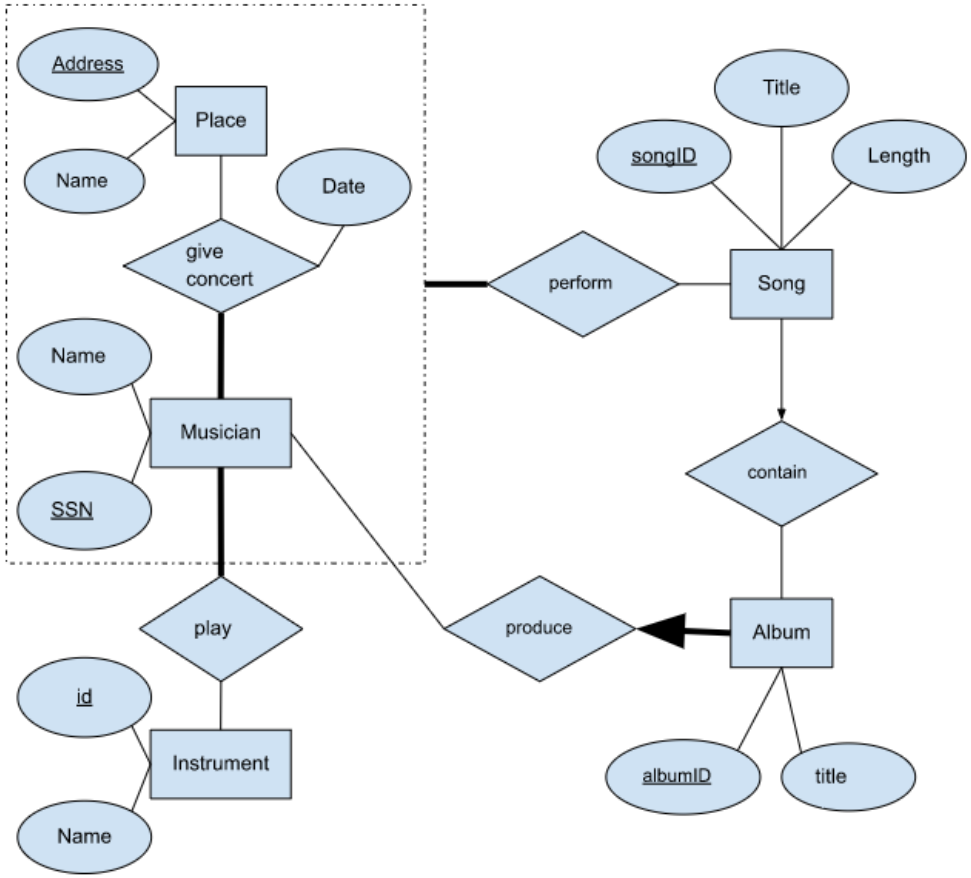
- Each <entity A> can *participate in relationships* with <any number / at least 1 / at most 1 / exactly 1 (according to the line that connects to entity A)> of <entity B> and <entity C> and ……
- Each nurse can care for any number of patients in any number of shifts
- Each shift has at least 1 nurse taking care of at least 1 patient
- Each patient has exactly 1 nurse taking care of him/her in exactly 1 shift

- HW1: This design enforces that every patient is taken care of in each shift.
  - NO!



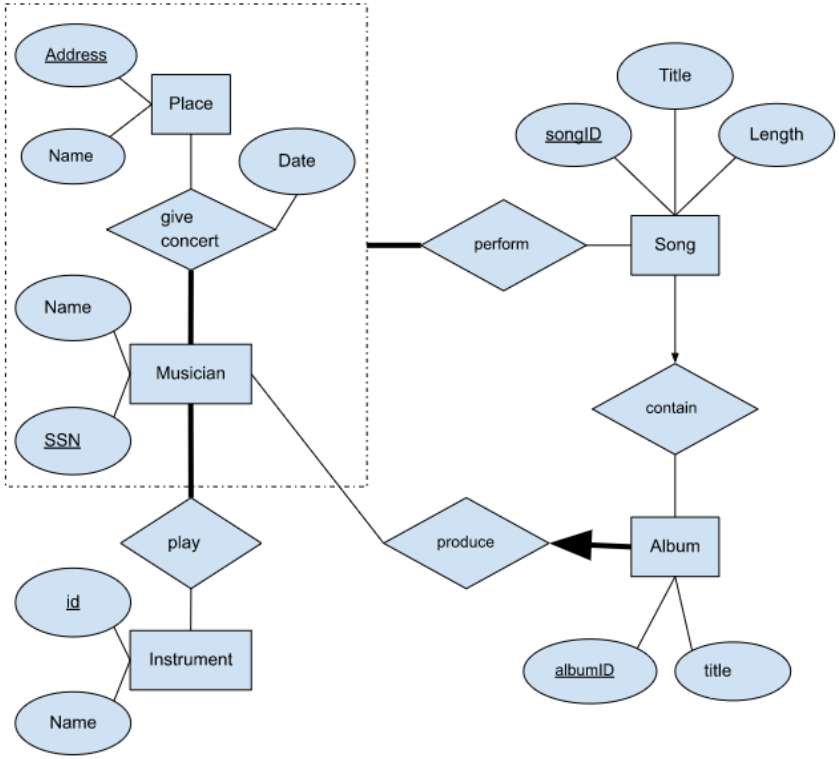
ER Diagram Example

- Does this design allow a musician to give multiple concerts in the same place on different dates?
  - NO!
  - Give_concert is identified as (address, SSN)
  - Date is not part of the key
- Does this design allow a musician to perform songs produced by another musician, in a concert?
  - NO!
  - ER Diagram can't enforce this



ER Diagram Example continued…

- Can a musician perform someone else's song?
  - YES!
- Can a concert involve no song performances?
  - NO!
- Can a musician perform two songs in from different albums on the same day?
  - YES!
- Can a musician perform the same song twice in the same place at different times?
  - NO!
- Can two musicians perform the same song at the same place?
  - YES!



Leetcode 175 (Easy) • Combine Two Tables - LeetCode

Table: `Person`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| personId    | int     |
| lastName    | varchar |
| firstName   | varchar |
+-------------+---------+
personId is the primary key
column for this table.
This table contains information
about the ID of some persons and
their first and last names.
```

Table: `Address`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| addressId   | int     |
| personId    | int     |
| city        | varchar |
| state       | varchar |
+-------------+---------+
addressId is the primary key
column for this table.
Each row of this table contains
information about the city and
state of one person with ID =
PersonId.
```

Write an SQL query to report the first name, last name, city, and state of each person in the `Person` table. If the address of a `personId` is not present in the `Address` table, report `null` instead.

Return the result table in any order.

Leetcode 175 Solution

```
SELECT FirstName, LastName, City, State
FROM Person JOIN Address
    ON Person.PersonId = Address.PersonId;
```

- WRONG! Why?
  - What if a person doesn't have an address?

```
SELECT FirstName, LastName, City, State
FROM Person LEFT JOIN Address
    ON Person.PersonId = Address.PersonId;
```

- LEFT JOIN
  - Want to keep records from the " person" table even the person does not have an address

Leetcode 183 (Easy)

[Customers Who Never Order - LeetCode]

Table: `Customers`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| id          | int     |
| name        | varchar |
+-------------+---------+
id is the primary key column for
this table.
Each row of this table indicates
the ID and name of a customer.
```

Table: `Orders`

```
+-------------+------+
| Column Name | Type |
+-------------+------+
| id          | int  |
| customerId  | int  |
+-------------+------+
id is the primary key column for
this table.
customerId is a foreign key of
the ID from the Customers table.
Each row of this table indicates
the ID of an order and the ID of
the customer who ordered it.
```

Write an SQL query to report all customers who never order anything. Include duplicate names is multiple customers of the same name do not order.

Return the result table in any order.

Leetcode 183 Solution

```
SELECT Name AS Customers
FROM Customers
MINUS
SELECT Customers.name
FROM Customers JOIN Orders
    ON Customers.id = Orders.customerId;
```

- WRONG! Why?
  - What if two customers of the same name both don't have orders?
  - What if one "Bob" orders and another "Bob" doesn't?

```
SELECT Name AS Customers
FROM Customers
WHERE id NOT IN (
    SELECT customerId
    FROM Orders
);
```

- Nested query
  - Find customerID for customers that ordered
  - Find the names of all other customers

Leetcode 596 (Easy)

- Classes More Than 5 Students - LeetCode

Table: `Courses`

```
+-------------+---------+
| Column Name | Type    |
+-------------+---------+
| student     | varchar |
| class       | varchar |
+-------------+---------+
(student, class) is the primary
key column for this table.
Each row of this table indicates
the name of a student and the
class in which they are enrolled.
```

Write an SQL query to report all the classes that have at least five students.

Return the result table in any order.

Leetcode 596 Solution

```
SELECT class FROM Courses
GROUP BY COUNT(student)
    HAVING COUNT(student) ⩾ 5;
```

- WRONG! Why?
  - Selection must be a subset of what you group by
  - You want to count the number of students in one group, however you didn't specify the group

```
SELECT class FROM Courses
GROUP BY class
    HAVING COUNT(*) ⩾ 5;
```

- Group by class
  - Now COUNT(*) counts the number of records inside each class
  - COUNT(*), COUNT(student), COUNT(DISTINCT student) don't matter in this case. Why?

Practice SQL Problem (Medium)

- Given the schema for tables “ workers” , “ company” , and “ employment”
- Workers table:
  - ssn INTEGER PRIMARY KEY
  - name VARCHAR2(30) NOT NULL
  - birthday DATE
- Company table:
  - name VARCHAR2(30) PRIMARY KEY
  - city VARCHAR2(30)
- Employment table:
  - ssn INTEGER
  - company_name VARCHAR2(30)
  - PRIMARY KEY(ssn, company_name)
- All VARCHAR2 stored in lower case

- Find the names and birthdays of
  - Workers who work in >= 2 cities
    OR
  - Workers who does not work for a company
- Except
  - Workers who work for Amazon or Microsoft

- Sort the result by descending alphabetical order of their name

- ***Notice pattern: (A OR B) AND !C
  - A UNION B MINUS C

Practice SQL Problem Solution Pt. 1

Find all workers who work in >= 2 cities

Query 1:

```
CREATE VIEW work_city_view AS
SELECT w.ssn
FROM workers w
    JOIN employment e ON w.ssn = e.ssn
    JOIN company c ON e.company_name = c.name
GROUP BY w.ssn
    HAVING COUNT(DISTINCT c.city) > 1
```

```
SELECT name, birthday
FROM workers
WHERE ssn IN work_city_view
```

Practice SQL Problem Solution Pt. 2

Find workers who do not work for a company

Query 2:

```
SELECT w.name, w.birthday
FROM workers w
WHERE ssn NOT IN (
    SELECT e.ssn FROM employment e
)
```

Find workers who work for Amazon or Microsoft

Query 3:

```
SELECT w.name, w.birthday
FROM workers w
    JOIN employment e ON w.ssn = e.ssn
WHERE e.company_name
    IN ('amazon', 'microsoft')
```

Solution: query1 UNION query2 MINUS query3

Practice SQL Problem Solution Complete

```
CREATE VIEW work_city_view AS
SELECT w.ssn
FROM workers w
    JOIN employment e ON w.ssn = e.ssn
    JOIN company c ON e.company_name = c.name
GROUP BY w.ssn
    HAVING COUNT(DISTINCT c.city) > 1;

SELECT name, birthday
FROM workers
WHERE ssn IN work_city_view
UNION
```

continued

```
SELECT w.name AS name, w.birthday AS birthday
FROM workers w
WHERE ssn NOT IN (
    SELECT e.ssn FROM employees e
)
MINUS
SELECT w.name, w.birthday
FROM workers w
    JOIN employment e
        ON w.ssn = e.ssn
WHERE e.company_name
    IN ('amazon', 'microsoft')
ORDER BY name DESC;
```

finish

Leetcode 262 (Hard)

- Trips and Users - LeetCode

Table: `Trips`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| id          | int      |
| client_id   | int      |
| driver_id   | int      |
| city_id     | int      |
| status      | enum     |
| request_at  | date     |
+-------------+----------+
id is the primary key for this table.
The table holds all taxi trips. Each trip has a
unique id, while client_id and driver_id are
foreign keys to the users_id at the Users table.
Status is an ENUM type of ('completed',
'cancelled_by_driver', 'cancelled_by_client').
```

Table: `Users`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| users_id    | int      |
| banned      | enum     |
| role        | enum     |
+-------------+----------+
users_id is the primary key for this table.
The table holds all users. Each user has a
unique users_id, and role is an ENUM type of
('client', 'driver', 'partner').
banned is an ENUM type of ('Yes', 'No').
```

The cancellation rate is computed by dividing the number of canceled (by client or driver) requests with unbanned users by the total number of requests with unbanned users on that day.

Write a SQL query to find the cancellation rate of requests with unbanned users (both client and driver must not be banned) each day between `"2013-10-01"` and `"2013-10-03"`. Round `Cancellation Rate` to two decimal points. You may NOT create views.

Return the result table in any order.

Hint: use `ROUND(A/B,2)` to round the quotient of A divided by B.

Solution

For each day:

- Find number of total requests, number of cancelled requests
- Check trip and user banned status

Check date

Since we want info for each day, group by date

```sql
SELECT
    request_at AS day,
    ROUND(cancelled / Total_requests, 2) AS "Cancellation Rate"
FROM
    (
        SELECT
            COUNT(*) as Total_requests,
            T.request_at,
            (
                SELECT
                    COUNT(T1.id)
                FROM
                    Trips T1
                    JOIN Users U1 ON T1.client_id = U1.users_id
                    AND U1.Banned = 'No'
                    JOIN Users U2 ON T1.driver_id = U2.users_id
                    AND U2.Banned = 'No'
                WHERE
                    T1.status <> 'completed'
                    AND T.request_at = T1.request_at
            ) AS cancelled
        FROM
            Trips T
            JOIN Users client ON T.client_id = client.users_id
            AND client.Banned = 'No'
            JOIN Users driver ON T.driver_id = driver.users_id
            AND driver.Banned = 'No'
        WHERE
            T.request_at BETWEEN '2013-10-01' AND '2013-10-03'
        GROUP BY
            T.request_at
    )
```

Leetcode 262 Create Tables

Table: `Trips`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| id          | int      |
| client_id   | int      |
| driver_id   | int      |
| city_id     | int      |
| status      | enum     |
| request_at  | date     |
+-------------+----------+
id is the primary key for this table.
The table holds all taxi trips. Each trip has a
unique id, while client_id and driver_id are
foreign keys to the users_id at the Users table.
Status is an ENUM type of ('completed',
'cancelled_by_driver', 'cancelled_by_client').
```

Table: `Users`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| users_id    | int      |
| banned      | enum     |
| role        | enum     |
+-------------+----------+
users_id is the primary key for this table.
The table holds all users. Each user has a
unique users_id, and role is an ENUM type of
('client', 'driver', 'partner').
banned is an ENUM type of ('Yes', 'No').
```

Create tables for the schema described.

Each trip MUST have a client, driver, status, and request date.

The same driver cannot drive the same client on the same day.

Each user must have a banned status and role.

Treat each ENUM attribute as a VARCHAR2.

Leetcode 262 Create Tables Solution (Trips)

Table: `Trips`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| id          | int      |
| client_id   | int      |
| driver_id   | int      |
| city_id     | int      |
| status      | enum     |
| request_at  | date     |
+-------------+----------+
id is the primary key for this table.
The table holds all taxi trips. Each trip has a
unique id, while client_id and driver_id are
foreign keys to the users_id at the Users table.
Status is an ENUM type of ('completed',
'cancelled_by_driver', 'cancelled_by_client').
```

```
CREATE TABLE Trips (
    id INTEGER PRIMARY KEY,
    client_id INTEGER NOT NULL,
    driver_id INTEGER NOT NULL,
    city_id INTEGER,
    status VARCHAR2(30) NOT NULL,
    request_at DATE NOT NULL,
    FOREIGN KEY (client_id) REFERENCES Users(users_id),
    FOREIGN KEY (driver_id) REFERENCES Users(users_id),
    CHECK (status IN ('completed',
                      'cancelled_by_driver',
                      'cancelled_by_client')),
    UNIQUE (client_id, driver_id, request_at)
)
```

Leetcode 262 Create Tables Solution (Users)

Table: `Users`

```
+-------------+----------+
| Column Name | Type     |
+-------------+----------+
| users_id    | int      |
| banned      | enum     |
| role        | enum     |
+-------------+----------+
users_id is the primary key for this table.
The table holds all users. Each user has a
unique users_id, and role is an ENUM type of
('client', 'driver', 'partner').
banned is an ENUM type of ('Yes', 'No').
```

```
CREATE TABLE Users (
    users_id INTEGER PRIMARY KEY,
    banned VARCHAR2(4) NOT NULL,
    role VARCHAR2(10) NOT NULL,
    CHECK (banned IN ('Yes', 'No')),
    CHECK (role IN ('client', 'driver', 'partner'))
)
```

Would the following insert statements work? (Assume empty table for each)

```
INSERT INTO Users VALUES (1, 'Yes', 'client');
```

YES!

```
INSERT INTO Users VALUES (15, 'YES', 'client');
```

NO!

```
INSERT INTO Users (users_id, banned) VALUES (1, 'No');
```

NO!

```
INSERT INTO Users (2, 'No', 'partner');
```

NO!