# CS 61:
# Database Systems

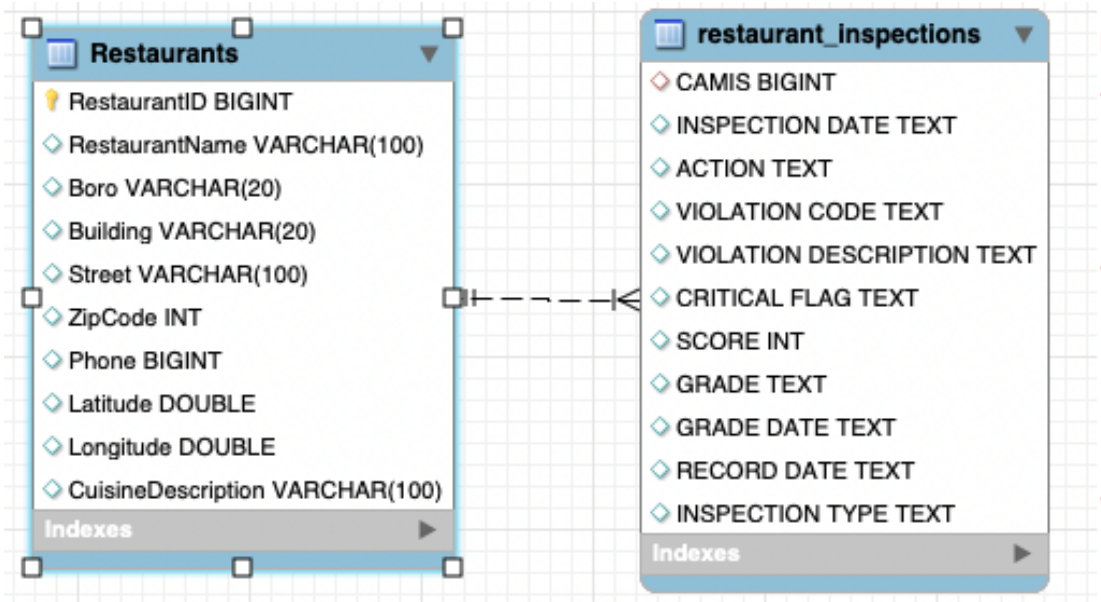
## Intermediate SQL



Adapted from Silberschatz, Korth, and Sundarshan unless otherwise noted
Augmented by Claude Opus

# Review: In our NYC health inspections schema, we left off with two tables

**We can tie Restaurants with their inspections using RestaurantID = CAMIS**




**Restaurants**

- **Each row is data about one restaurant**
- **Can change name, cuisine, or move to new address -> just update one row in this table**

**restaurant_inspections**

- **Each row is data about one violation in one inspection of a restaurant**
- **We removed the duplicate information about the restaurant's name, address, cuisine**
- **We just keep the CAMIS (ID) integer for each restaurant in this inspections table**

# Agenda

1. Views
2. Object-oriented data
3. Generated columns
4. Time variant data

# Views create virtual tables based on underlying database tables

**Views**

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database)
- Consider a person who needs to know an instructor's name and department, but not the salary. This person should see a relation described, in SQL, by

  **SELECT** *ID*, *name*, *dept_name*
  **FROM** *instructor*

- A **view** can provide a mechanism to hide certain data from the view of certain users

Format

**CREATE VIEW** v **AS**
**SELECT** $A_1$, $A_2$, ..., $A_n$
**FROM** r

Then
**SELECT** * **FROM** v

- **Why not just save data into a table?**
- **Database saves query but not data!**
- **That way if data in underlying tables changes, view is automatically up to date**
- **Can then query the view (or define other views) as if it were a relation**
- **An alternative is a Materialized View where data is actually stored, but MySQL does not support them**

# Views can sometimes be used to modify underlying table

A SQL view can update (INSERT, UPDATE, DELETE) the underlying base table when **all** of the following conditions are met:

**1. Single Base Table**
The view must reference exactly **one** base table. Views built on joins across multiple tables are generally not updatable

**2. No Aggregate Functions**
The view cannot use aggregate functions (e.g., SUM(), COUNT(), AVG(), MIN(), MAX)

**3. No GROUP BY or HAVING**
The view cannot group rows together since there is no way to map a grouped result back to individual rows

**4. No DISTINCT**
The view cannot use DISTINCT since it eliminates duplicate rows and the database cannot determine which original row to update

**5. No Generated / Derived Columns**
The view should not contain computed expressions like Price * Quantity or CONCAT(Firstname, ', ', LastName) AS Name since these do not map directly to a single column in the base table

**6. No Set Operations**
The view cannot use UNION, INTERSECT, or EXCEPT since rows from multiple queries cannot be traced back to a single source

**7. All NOT NULL Columns Included (for INSERT)**
When inserting through a view, all columns in the base table that are NOT NULL and lack a default value must be included in the view

# Practice

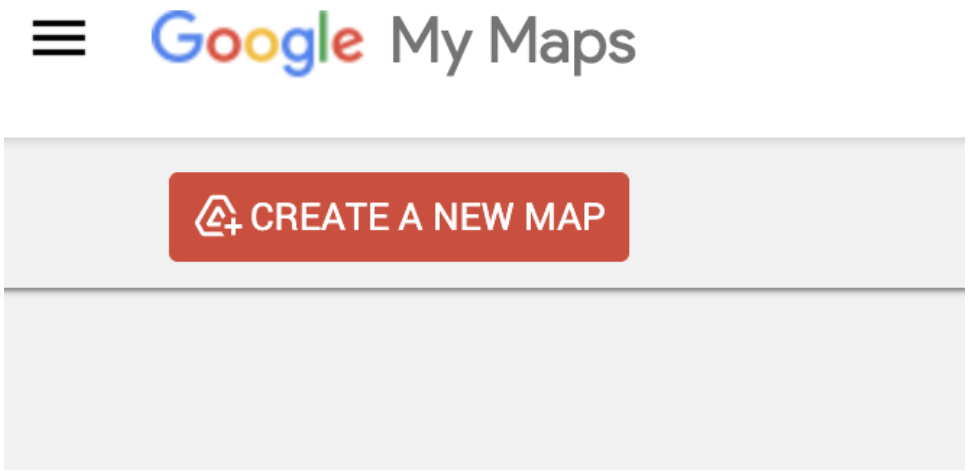
**use nyc_data;**

Imagine you are creating a web site that provides health inspection results for cuisine LIKE %fruit% restaurants in Manhattan. Each restaurant is displayed on a map and shows the average health inspection score. Create a view for this map-based data:

- You are focused on Manhattan fruit and vegetable restaurants only
- You don't need all the info in the Restaurants and Inspections tables
- Make a view for your map-based website called MapData that has attributes:
  - RestaurantID
  - RestaurantName
  - Latitude
  - Longitude
  - Average health inspection score (NULL if no inspections for this restaurant)
  - Days since last inspection (use DATEDIFF, CURDATE, MAX(InspectionDate)
- Make sure you list the new restaurant you created yesterday (Tim's Tasty Treats) listed, even though it has no health inspection reports yet (use NULL avg score, or for more challenge, make NULLs zero – Hint: use IF or COALESCE)
- Check your view works by querying it with a SELECT statement as if your view were a table

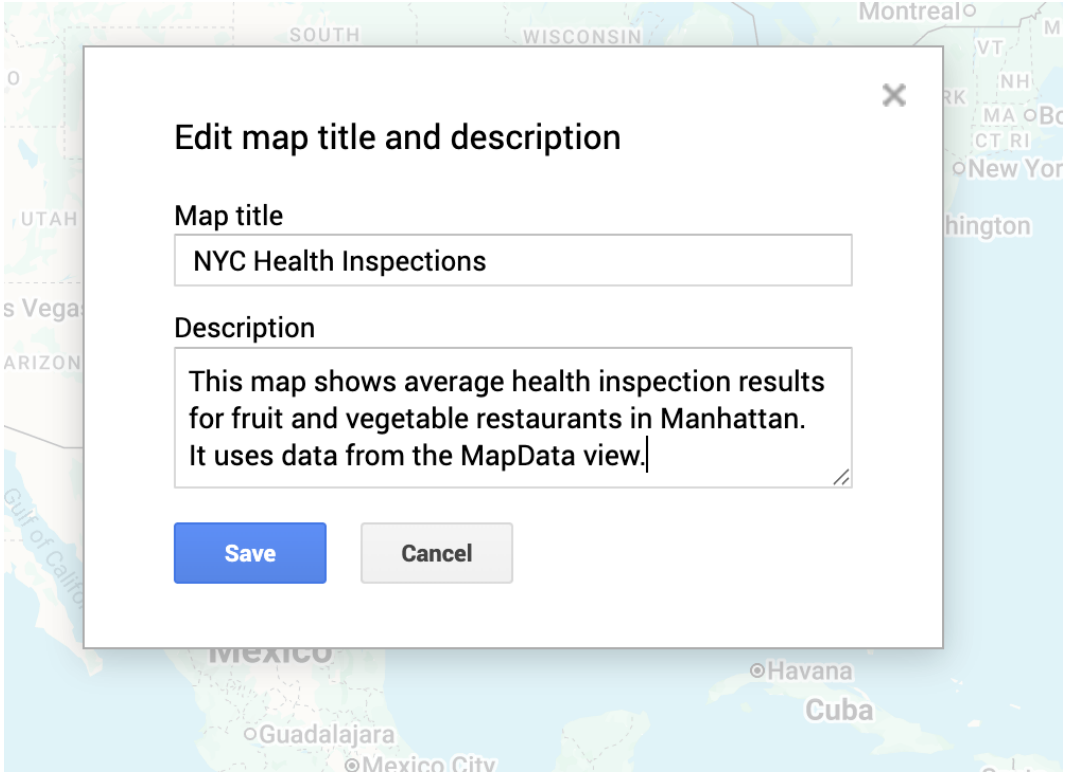
# Plot your data on a map

1. Go to *https://www.google.com/mymaps*
2. Create new map



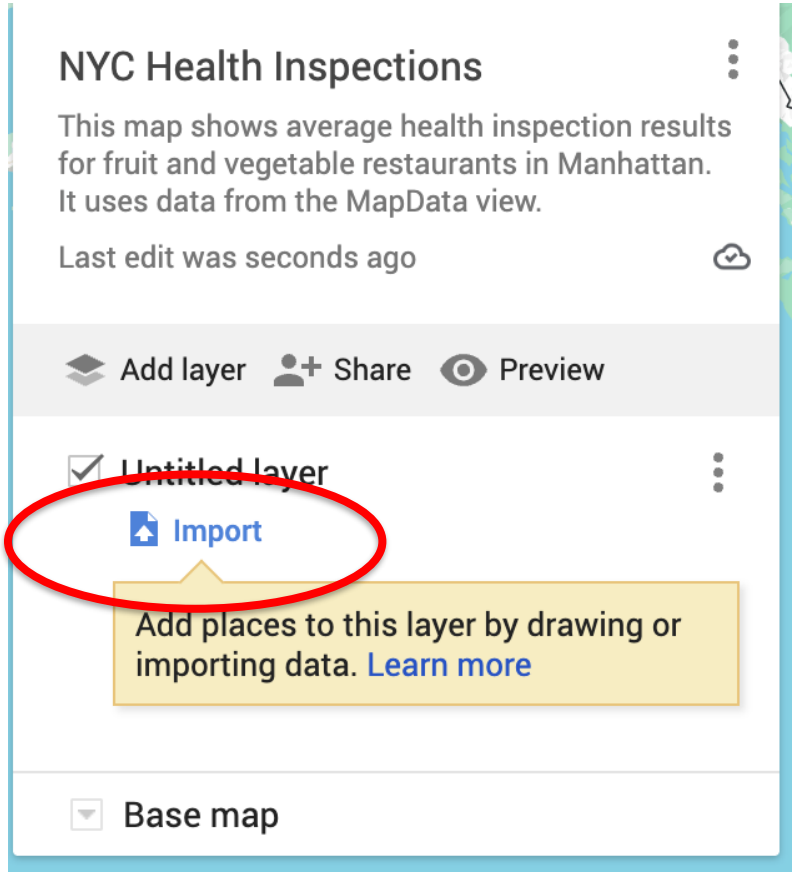
# Plot your data on a map

1. Go to *https://www.google.com/mymaps*
2. Create new map
3. Give new map a title and description then Save



# Plot your data on a map

1. Go to *https://www.google.com/mymaps*
2. Create new map
3. Give new map a title and description then Save
4. Click Import under layer



# Plot your data on a map

1. Go to [https://www.google.com/mymaps](https://www.google.com/mymaps)
2. Create new map
3. Give new map a title and description then Save
4. Click Import under layer
5. Export data from MapData view as .csv (I called mine day6_fruitveg_map.csv)

```
36  DROP VIEW IF EXISTS MapData;
37  CREATE VIEW MapData AS
38      SELECT RestaurantID, RestaurantName, Building, Street, ZipCode, Latitude, Longitude, COALESCE(AVG(score),0) AS AvgScore
39      FROM restaurants r LEFT JOIN restaurant_inspections i ON r.RestaurantID = i.CAMIS
40      WHERE boro = 'Manhattan' AND CuisineDescription LIKE '%fruit%'
41      GROUP BY RestaurantID;
42
43  SELECT * from MapData;
44
45  -- export results to .csv and import into Google Maps
46  -- see instructions https://www.google.com/earth/outreach/learn/visualize-your-data-on-a-custom-map-using-google-my-maps/
```

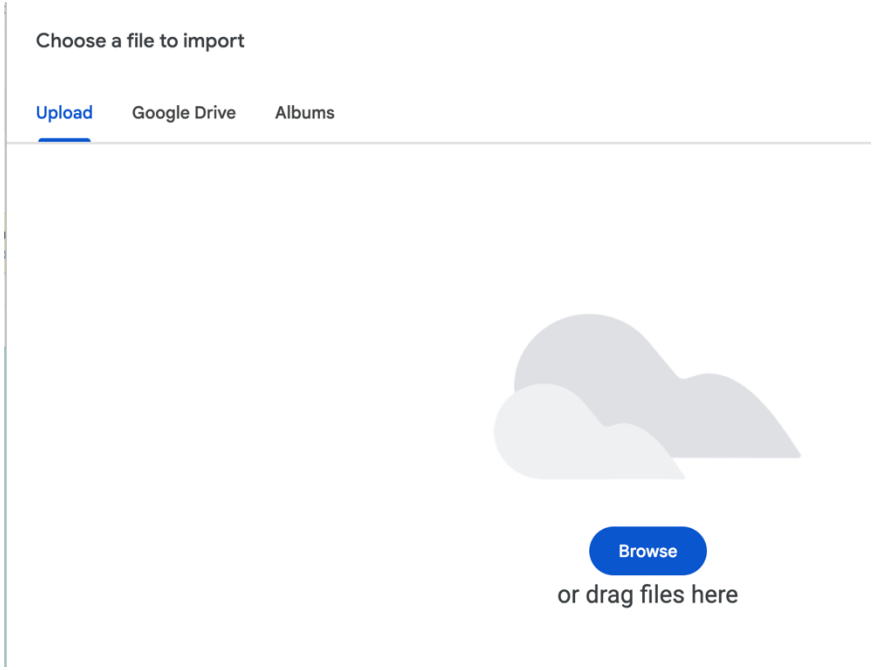
00% 19:43

Result Grid Filter Rows: Search Export:

| Restaura... | RestaurantName | Building | Street | ZipCode | Latitude | Longitude | AvgScore |
|---|---|---|---|---|---|---|---|
| 1111 | Tim's Tasty Treats | NULL | NULL | NULL | NULL | NULL | 0.0000 |
| 40698788 | JUICE GENERATION | 644 | NINTH AVENUE STREET LEVEL | 10036 | 40.76052767318 | -73.99108392015 | 7.2500 |
| 40735129 | JUICY LUCY | 85 | AVENUE A | 10009 | 40.72516615467 | -73.98419050577 | 25.4000 |
| 40881627 | JUICE GENERATION | 117 | WEST 72 STREET | 10023 | 40.77766717208 | -73.97939022913 | 20.5000 |
| 41333632 | JUICE BAR PLUS | 700 | 3 AVENUE | 10017 | 40.75196956292 | -73.97358004522 | 9.0000 |
| 41481494 | JACK STUDIOS | 601 | WEST 26 STREET | 10001 | 40.75092669619 | -74.005955224 | 9.2500 |
| 41562131 | HAWA SMOOTHIES | 181 | EAST BROADWAY | 10002 | 40.71412728399 | -73.98923964386 | 29.4000 |
| 41585053 | REGGAE SUN DELIGHTS NATURAL JUICE BAR | | WEST 145 STREET | NULL | 0 | 0 | 21.6875 |
| 41667090 | COOL FRESH JUICE BAR | 2661 | BROADWAY | 10025 | 40.79797883963 | -73.96936120543 | 15.2857 |
| 41699397 | JUICY CUBE | 674 | LEXINGTON AVENUE | 10022 | 40.76011231429 | -73.96973244418 | 23.6500 |

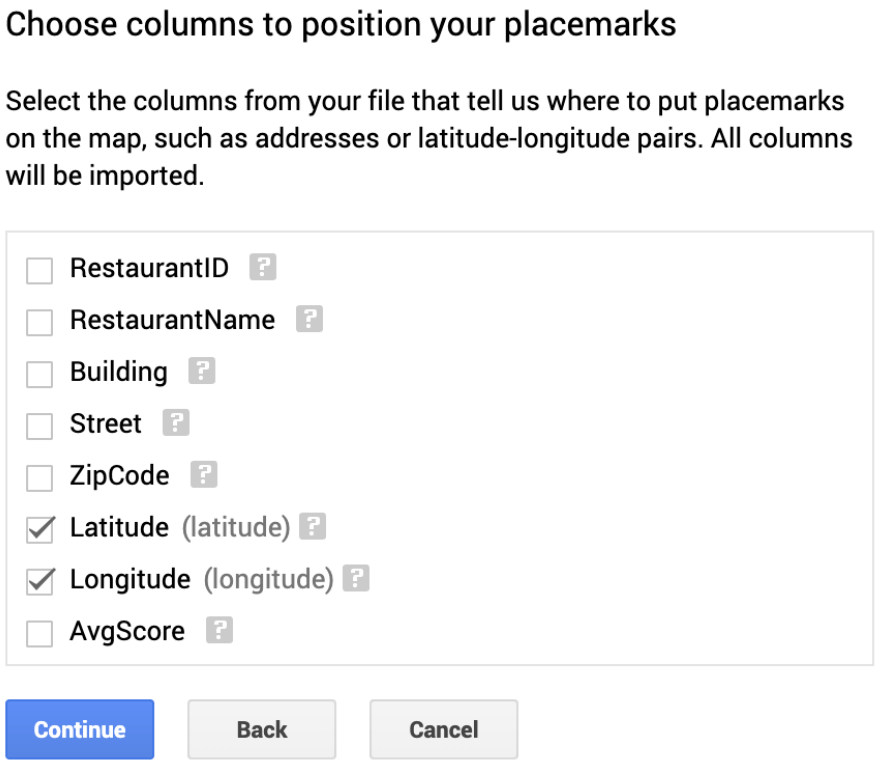
# Plot your data on a map

1. Go to *https://www.google.com/mymaps*
2. Create new map
3. Give new map a title and description then Save
4. Click Import under layer
5. Export data from MapData view as .csv (I called mine day6_fruitveg_map.csv)
6. Drop file onto Google Maps



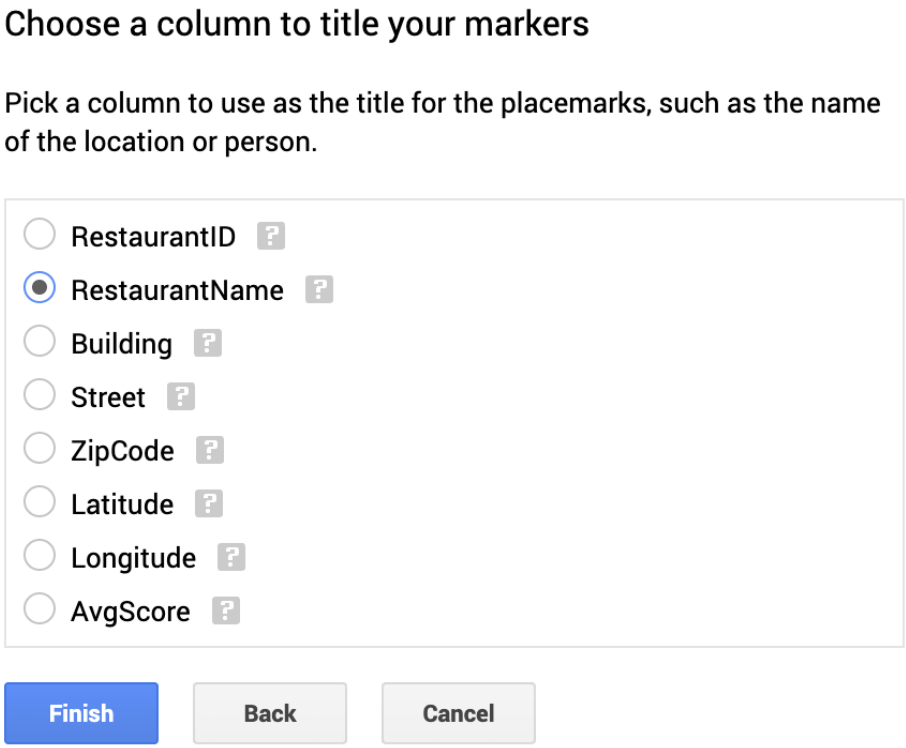
# Plot your data on a map

1. Go to *https://www.google.com/mymaps*
2. Create new map
3. Give new map a title and description then Save
4. Click Import under layer
5. Export data from MapData view as .csv (I called mine day6_fruitveg_map.csv)
6. Drop file onto Google Maps
7. Choose columns for lat/long



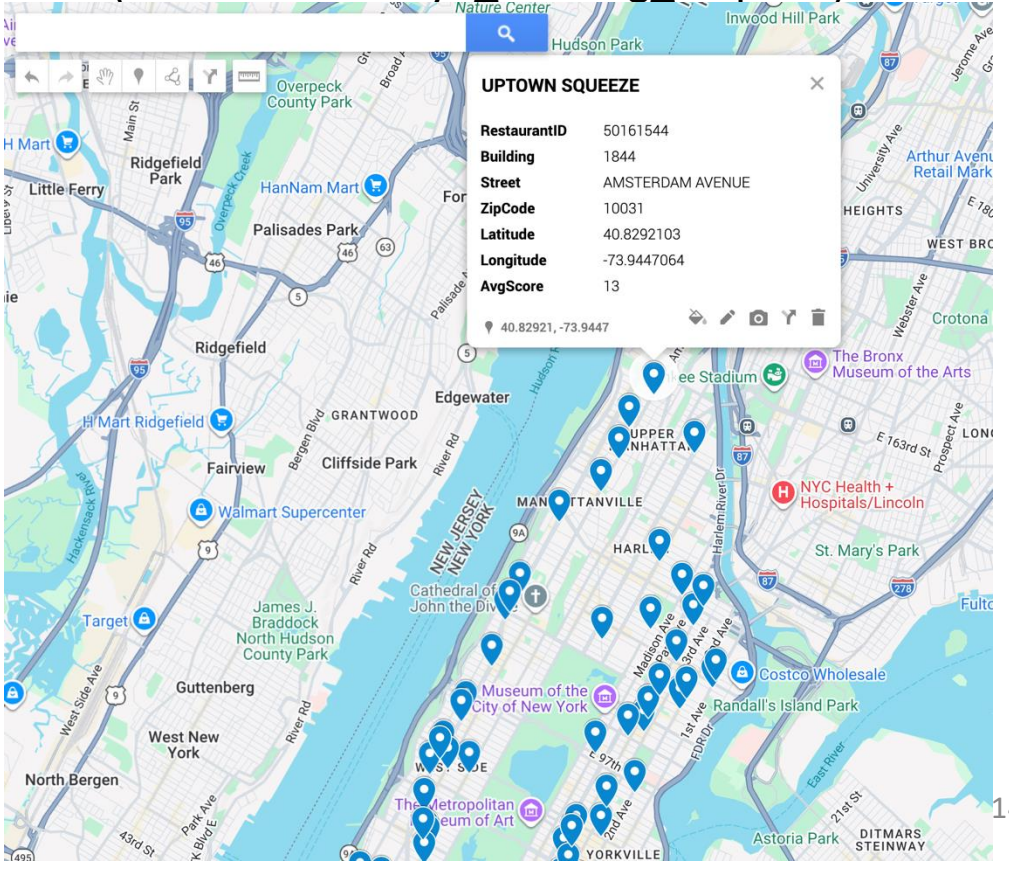
# Plot your data on a map

1. Go to [https://www.google.com/mymaps](https://www.google.com/mymaps)
2. Create new map
3. Give new map a title and description then Save
4. Click Import under layer
5. Export data from MapData view as .csv (I called mine day6_fruitveg_map.csv)
6. Drop file onto Google Maps
7. Choose columns for lat/long
8. Choose title for pins



# Plot your data on a map

1. Go to [https://www.google.com/mymaps](https://www.google.com/mymaps)
2. Create new map
3. Give new map a title and description then Save
4. Click Import under layer
5. Export data from MapData view as .csv (I called mine day6_fruitveg_map.csv)
6. Drop file onto Google Maps
7. Choose columns for lat/long
8. Choose title for pins
9. Zoom in to Manhattan and click pin

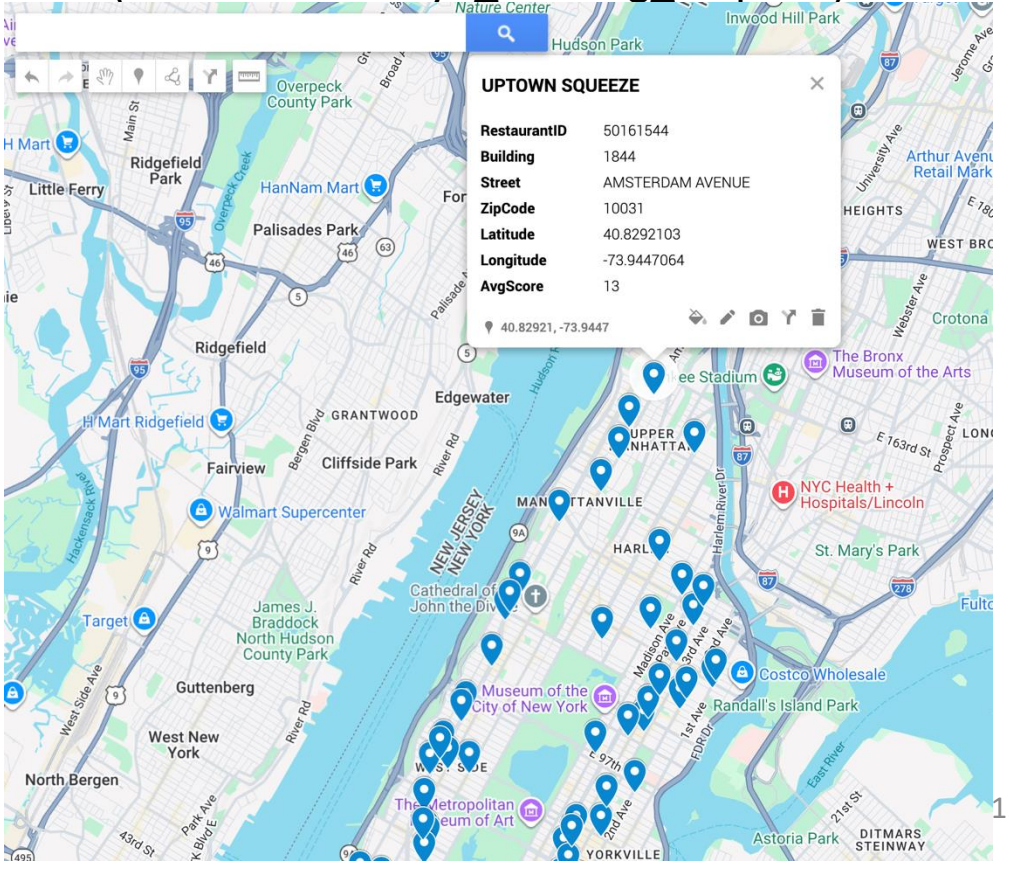


# Plot your data on a map

1. Go to [https://www.google.com/mymaps](https://www.google.com/mymaps)
2. Create new map
3. Give new map a title and description then Save
4. Click Import under layer
5. Export data from MapData view as .csv (I called mine day6_fruitveg_map.csv)
6. Drop file onto Google Maps
7. Choose columns for lat/long
8. Choose title for pins
9. Zoom in to Manhattan and click pin
10. Refresh data with

    SELECT * FROM MapData;

Then must re-drop data onto Map here but your own app could refresh itself.



# Agenda

1. Views
2. Object-oriented data
3. Generated columns
4. Time variant data

# Object-oriented data

DartAir airline has employees who are either:

**Pilots**

- License type
- Flight hours
- Medical type

**Mechanics**

- Certification
- Last safety training

**Accountants**

- Title
- CPA date

**Other**

All employees have common attributes:

- First name, last name, middle initial, date of hire

Each type of employee (other than 'other') have additional job-related attributes as shown above

# We *could* create one large table and use NULL for irrelevant attributes

```
CREATE TABLE EmployeesBigTable (
-- Common attributes
  EmployeeID int NOT NULL AUTO_INCREMENT,
  FullName varchar(100) NOT NULL,
  Salary decimal(10,2) DEFAULT NULL,
  HireDate date DEFAULT NULL,
  EmployeeType enum('Pilot','Mechanic','Accountant','Other') NOT NULL,
-- Pilot attributes
  LicenseType varchar(50) DEFAULT NULL,
  FlightHours int DEFAULT '0',
  MedicalType varchar(50) DEFAULT NULL,
-- Mechanic attributes
  CertificationType varchar(50) DEFAULT NULL,
  LastSafetyTraining date DEFAULT NULL,
-- Accountant attributes
  Title varchar(50) DEFAULT NULL,
  CPADate date DEFAULT NULL,
  PRIMARY KEY (EmployeeID))
```

**EmployeeType is an enumeration of 'Pilot', 'Mechanic', 'Accountant', or 'Other')**

**Use DEFAULT NULL (or maybe 0) for attributes that may not pertain to an employee category**

# We *could* create one large table and use NULL for irrelevant attributes

**Another approach is to create separate tables for each employee category and just repeat the base class attributes in each table**

**Also gross, but people do that too…**
**Why?**

**INSERT INTO** EmployeesBigTable (FullName, Salary, HireDate, EmployeeType, LicenseType, FlightHours, MedicalType)
**VALUES** ('Chuck Yeager', 100000, '1960-6-1', 'Pilot', 'ATP', '5000', 'III'); -- Pilot

**INSERT INTO** EmployeesBigTable (FullName, Salary, HireDate, EmployeeType, CertificationType, LastSafetyTraining)
**VALUES** ('Snuffy Smith', 70000, '1992-7-15', 'Mechanic', 'A&P', '2026-01-02'); -- Mechanic

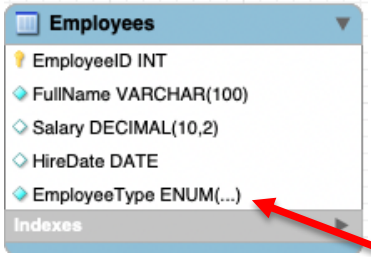
**INSERT INTO** EmployeesBigTable (FullName, Salary, HireDate, EmployeeType, Title, CPADate)
**VALUES** ('Scrooge McDuck', 350000, '2023-5-17', 'Accountant', 'CPA Accountant', '2021-10-11'); -- Accountant

**INSERT INTO** EmployeesBigTable (FullName, Salary, HireDate, EmployeeType)
**VALUES** ('Nobody Special', 45000, '2026-2-27', 'Other'); -- Other

**Lots of NULLs**
**Kind of gross, but people do this!**

| EmployeeID | FullName | Salary | HireDate | EmployeeType | LicenseType | FlightHours | MedicalType | CertificationTy... | LastSafetyTraining | Title | CPADate |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Chuck Yeager | 100000.00 | 1960-06-01 | Pilot | ATP | 5000 | III | NULL | NULL | NULL | NULL |
| 2 | Snuffy Smith | 70000.00 | 1992-07-15 | Mechanic | NULL | 0 | NULL | A&P | 2026-01-02 | NULL | NULL |
| 3 | Scrooge McDuck | 350000.00 | 2023-05-17 | Accountant | NULL | 0 | NULL | NULL | NULL | CPA Accountant | 2021-10-11 |
| 4 | Nobody Special | 45000.00 | 2026-02-27 | Other | NULL | 0 | NULL | NULL | NULL | NULL | NULL |

# Another idea: Employee base table plus specialty tables

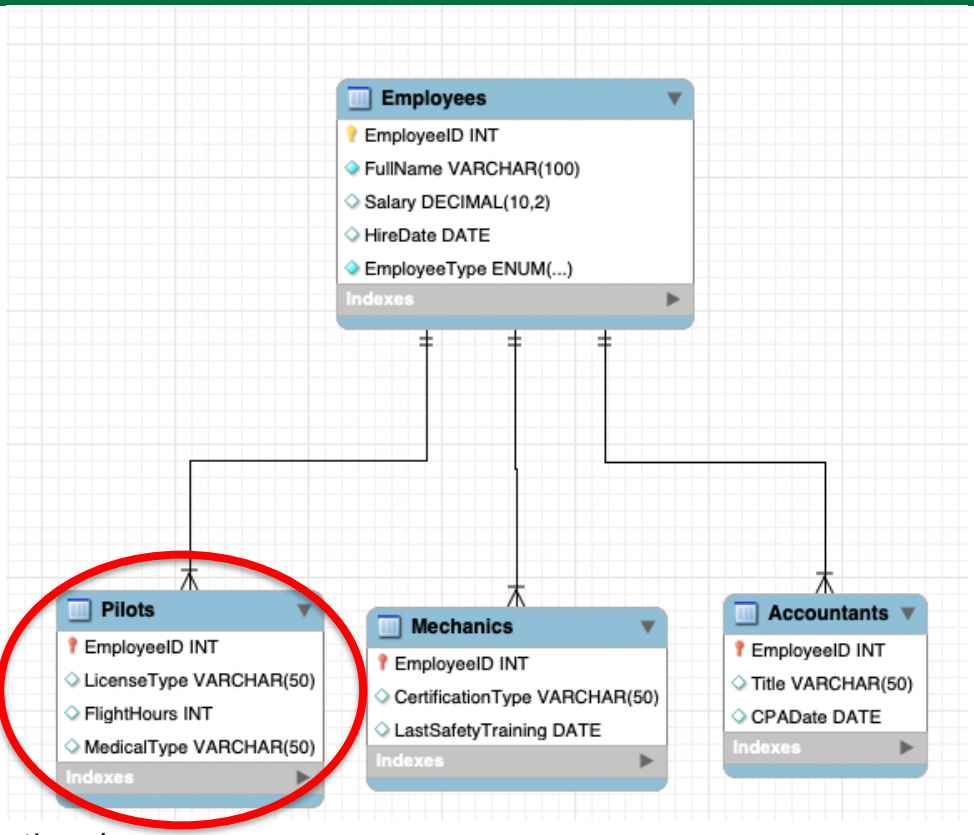



**All Employees have these attributes**

**EmployeeType is an enumeration of 'Pilot', 'Mechanic','Accountant', or 'Other')**

```
CREATE TABLE Employees (
  EmployeeID INT NOT NULL AUTO_INCREMENT,
  FullName VARCHAR(100) NOT NULL,
  Salary DECIMAL(10,2) DEFAULT NULL,
  HireDate DATE DEFAULT NULL,
  EmployeeType ENUM('Pilot','Mechanic','Accountant','Other') NOT NULL DEFAULT 'Other',
  PRIMARY KEY (EmployeeID)
)
```

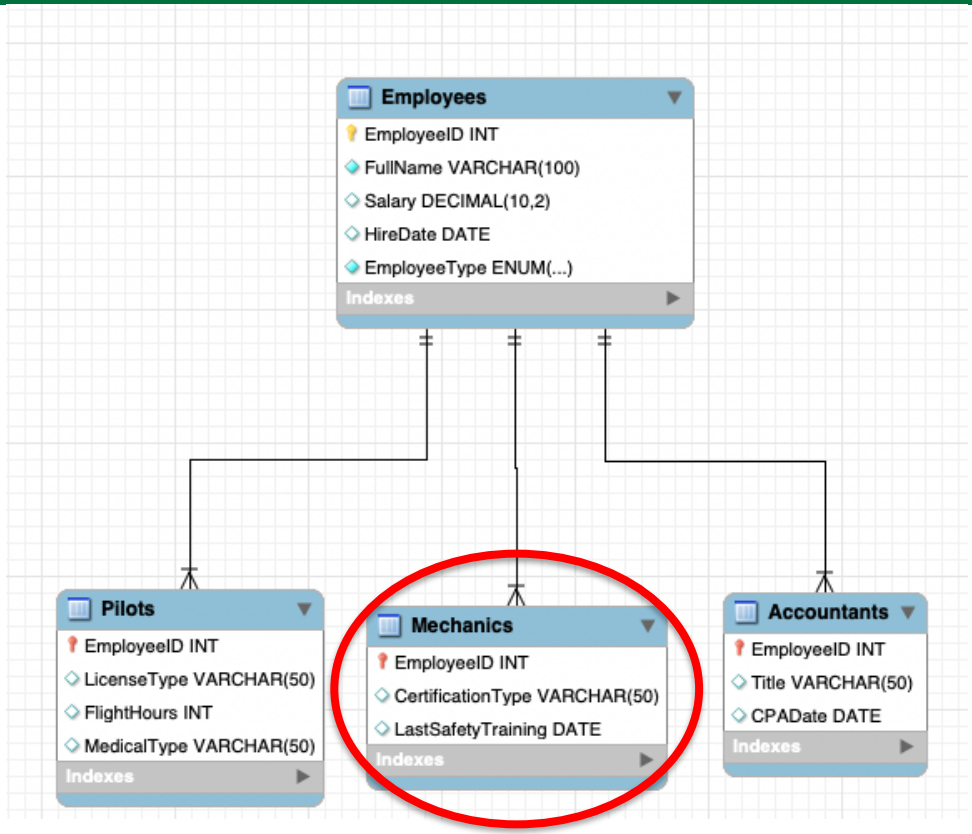
# Pilot table has specialty info relevant to pilots (not others)




```
CREATE TABLE Pilots (
  EmployeeID INT NOT NULL,
  LicenseType VARCHAR(50) DEFAULT NULL,
  FlightHours INT DEFAULT '0',
  MedicalType VARCHAR(50) DEFAULT NULL,
  PRIMARY KEY (EmployeeID),
  CONSTRAINT `pilots_ibfk_1` FOREIGN KEY (EmployeeID) REFERENCES Employees (EmployeeID)
        ON DELETE CASCADE)
```

**Set EmployeeID to be a foreign key on Employees.EmployeeID**

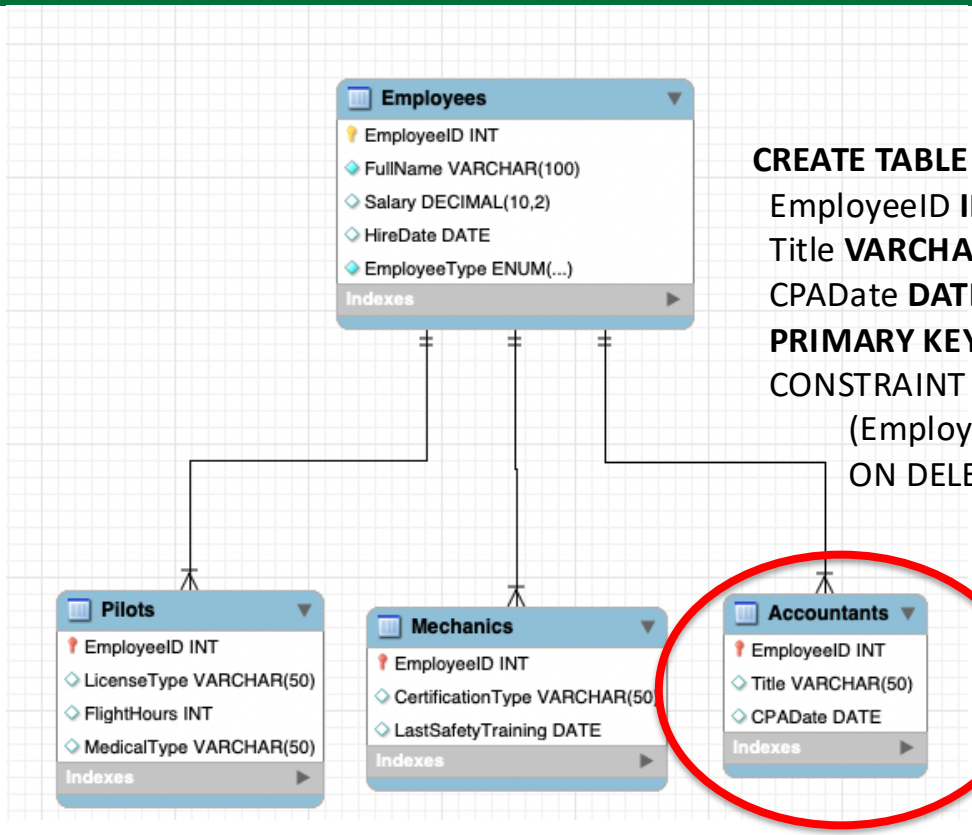
# Mechanics table has specialty info relevant to mechanics (not others)




```
CREATE TABLE Mechanics (
  EmployeeID INT NOT NULL,
  CertificationType VARCHAR(50) DEFAULT NULL,
  LastSafetyTraining DATE DEFAULT NULL,
  PRIMARY KEY (EmployeeID),
  CONSTRAINT `mechanics_ibfk_1` FOREIGN KEY (EmployeeID) REFERENCES Employees (EmployeeID)
        ON DELETE CASCADE)
```

**Set EmployeeID to be a foreign key on Employees.EmployeeID**

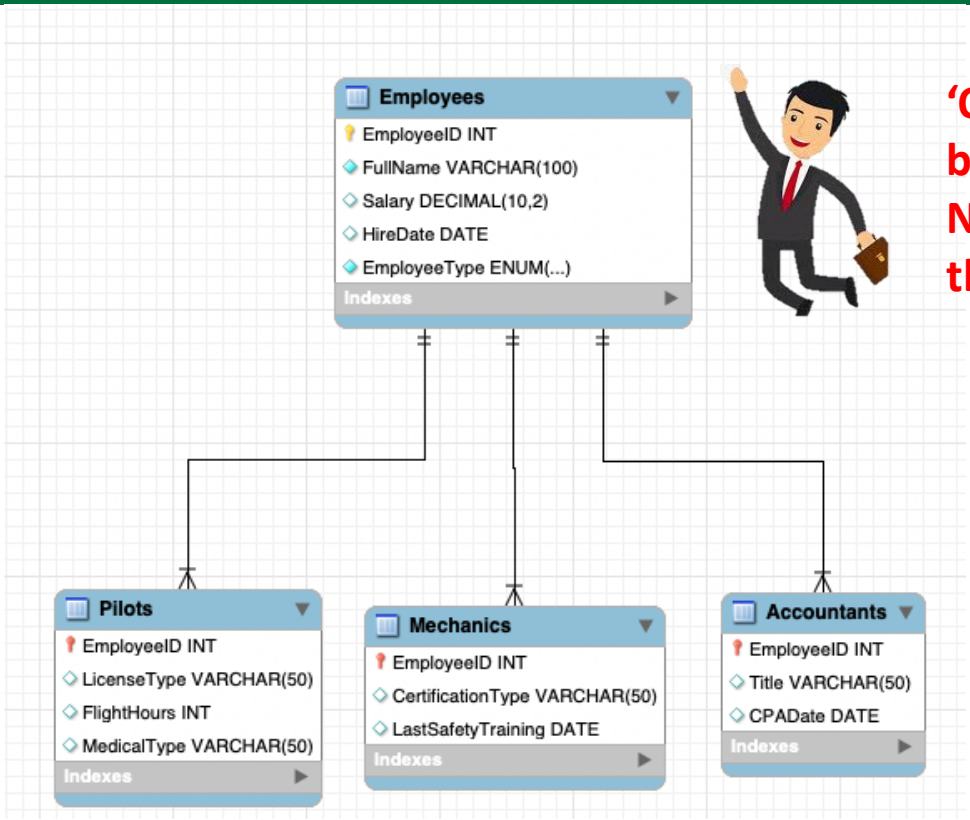
# Accountants table has specialty info relevant to accountants (not others)




**CREATE TABLE** Accountants (
EmployeeID **INT NOT NULL**,
Title **VARCHAR**(50) **DEFAULT NULL**,
CPADate **DATE DEFAULT NULL**,
**PRIMARY KEY** (EmployeeID),
CONSTRAINT `accountants_ibfk_1` FOREIGN KEY
(EmployeeID) REFERENCES Employees (EmployeeID)
ON DELETE CASCADE)

**Set EmployeeID to be a foreign key on Employees.EmployeeID**

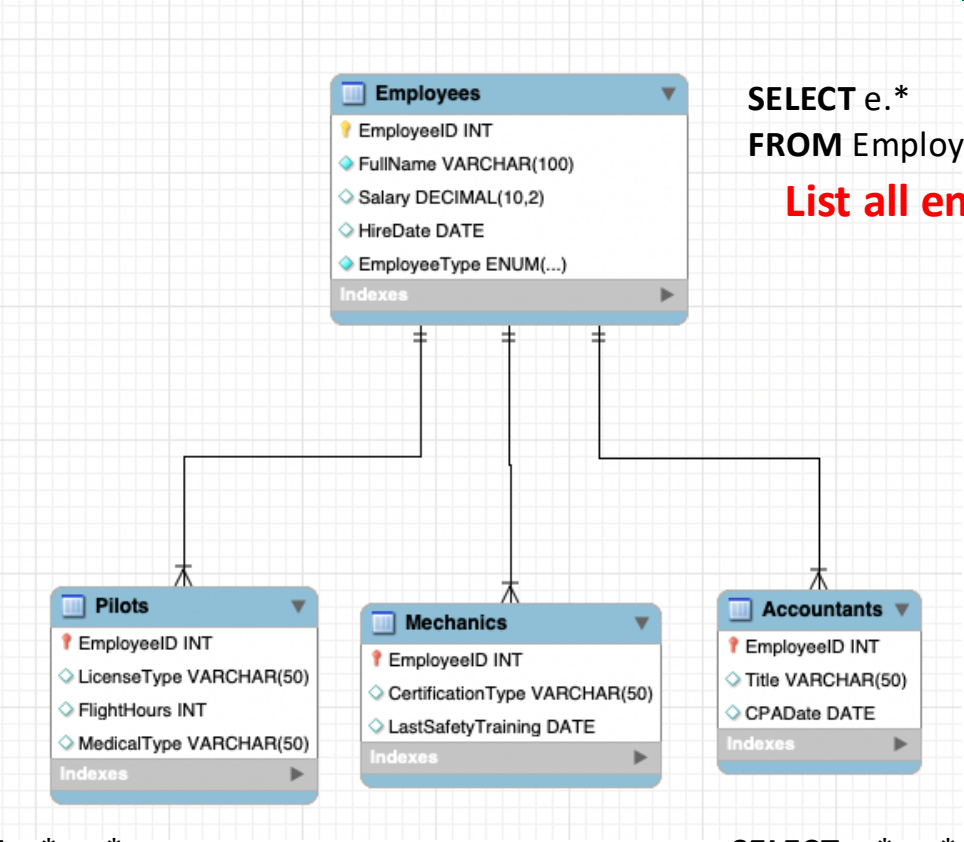
# What should we do about the ‘Other’ category?






**‘Other’ category can use the base table**
**No need for another table that has no extra attributes**

# JOIN specialty tables to base table to get details for employee categories




**SELECT** e.*
**FROM** Employees e

**List all employees**

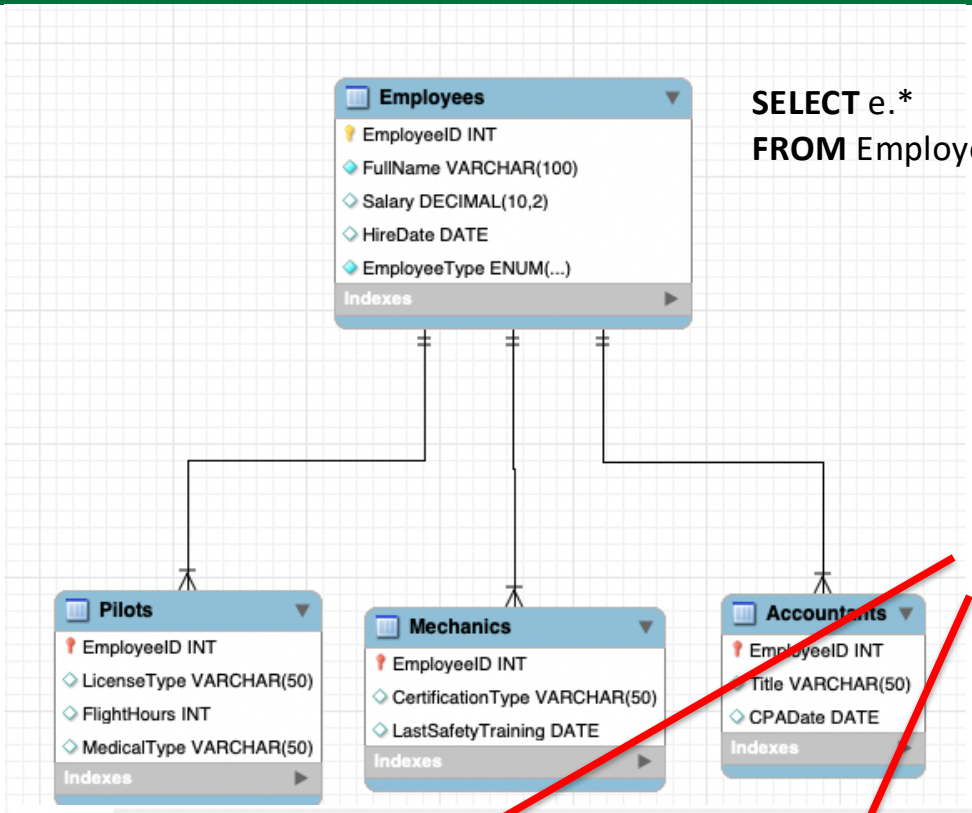
**SELECT** e.*, p.*
**FROM** Employees e
**JOIN** Pilots p **USING** (EmployeeID);

**SELECT** e.*, a.*
**FROM** Employees e
**JOIN** Accountants a USING (EmployeeID);

**List specialty details**

**SELECT** e.*, m.*
**FROM** Employees e
**JOIN** Mechanics m USING (EmployeeID);

# Employee base table plus specialty tables for pilots, mechanics, accountants




**SELECT** e.*
**FROM** Employees e

**How can we get rid of EmployeeID appearing twice? Don't use SELECT ***

```
51 •  SELECT e.*, p.*
52    FROM Employees e JOIN Pilots p USING (EmployeeID);
53
```

00% 12:51

Result Grid Filter Rows: Search Export:

| EmployeeID | FullName | Salary | HireDate | EmployeeType | EmployeeID | LicenseType | FlightHours | MedicalType |
|---|---|---|---|---|---|---|---|---|
| 1 | Chuck Yeager | 100000.00 | 1960-06-01 | Pilot | 1 | ATP | 5000 | III |

**Common attributes** **Pilot attributes**

# Practice

Run day6_dartair.sql from today's web page to create tables for base employees, pilots, mechanics, and accountants

- Add a new employee in a two-step process:
  1. Insert new employee into Employee table
  2. Then insert into appropriate specialty table (pilot, mechanic, accountant, other)
     When inserting into a specialty table, you'll need the EmployeeID of the last insert
        Could query for MAX EmployeeID of the base table (and hope others don't insert between your queries!)
        or better,
        Can use LAST_INSERT_ID() – session safe!
- Create a view for Pilots that returns all employee and pilot-specific attributes, test your view by querying for all Pilots

# One more approach: use JSON for category-specific attributes

```
CREATE TABLE Employees2 (
    EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
    FullName VARCHAR(100) NOT NULL,
    Salary DECIMAL(10, 2),
    HireDate DATE,
    EmployeeType ENUM('Pilot', 'Mechanic', 'Accountant', 'Other') NOT NULL,
    Properties JSON DEFAULT NULL
);
```

**Store base attributes as usual**

**Add field for JSON name:value text to store category-specific attributes such as FlightHours for pilots or CPADate for accountants**

# Insert category-specific data in a JSON name:value pair string

```
CREATE TABLE Employees2 (
    EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
    FullName VARCHAR(100) NOT NULL,
    Salary DECIMAL(10, 2),
    HireDate DATE,
    EmployeeType ENUM('Pilot', 'Mechanic', 'Accountant', 'Other') NOT NULL,
    Properties JSON DEFAULT NULL
);
```

**We will see at the end of the term that NoSQL database work like this!**

**Store base attributes as usual**

**Use JSON name:value pairs for other category-specific attributes in JSON field (called Properties here)**

```
-- insert pilot
INSERT INTO Employees2 (FullName, Salary, HireDate, EmployeeType, Properties)
VALUES ("Ace Pilot", 110000, '2026-4-11', 'Pilot','{"LicenseType":"ATP", "FlightHours":2222, "MedicalType":"II"}');
-- insert mechanic
INSERT INTO Employees2 (FullName, Salary, HireDate, EmployeeType, Properties)
VALUES("Joe Mechanic", 77000, '2026-4-2','Mechanic','{"CertificationType":"A&P", "LastSafetyTraining":"2026-4-2"}');
-- insert accountant
INSERT INTO Employees2 (FullName, Salary, HireDate, EmployeeType, Properties)
VALUES("Alice Accountant", 123000, '2026-4-5','Accountant','{"Title":"CPA Accountant", "CPADate":"2026-4-5"}');
-- insert other
INSERT INTO Employees2 (FullName, Salary, HireDate, EmployeeType, Properties)
VALUES("Random Dude", 33000, '2026-4-6','Other');

-- check that it worked, should see JSON attribute called Properties
SELECT * FROM Employees2;
```

# Retrieve category-specific data with JSON_EXTRACT

```
CREATE TABLE Employees2 (
    EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
    FullName VARCHAR(100) NOT NULL,
    Salary DECIMAL(10, 2),
    HireDate DATE,
    EmployeeType ENUM('Pilot', 'Mechanic', 'Accountant', 'Other') NOT NULL,
    Properties JSON DEFAULT NULL
);
```

Retrieve base attributes as usual

Extract JSON name:value pairs from Properties attribute using JSON_EXTRACT

```
-- insert pilot
INSERT INTO Employees2 (FullName, Salary, HireDate, EmployeeType, Properties)
VALUES ("Ace Pilot", 110000, '2026-4-11', 'Pilot','{"LicenseType":"ATP", "FlightHours":2222, "MedicalType":"II"}');
```

```
250     -- get pilots
251 •   SELECT FullName, Salary, HireDate, EmployeeType,
252         JSON_EXTRACT(Properties, '$.FlightHours') AS FlightHours,
253         JSON_EXTRACT(Properties, '$.LicenseType') AS LicenseType,
254         JSON_UNQUOTE(JSON_EXTRACT(Properties, '$.MedicalType')) AS MedicalType, -- remove quotes
255         JSON_EXTRACT(Properties, '$.NonExistentField') AS NonExistentField -- set to NULL
256         FROM Employees2
257         WHERE EmployeeType = 'Pilot';
258
```

Use UNQUOTE to remove quotes

Non-existent fields returned as NULL

100% 11:252

Result Grid Filter Rows: Search Export:

| FullName | Salary | HireDate | EmployeeType | FlightHours | LicenseType | MedicalType | NonExistentField |
|---|---|---|---|---|---|---|---|
| Ace Pilot | 110000.00 | 2026-04-11 | Pilot | 2222 | "ATP" | II | NULL |

# Agenda

1. Views
2. Object-oriented data
3. Generated columns
4. Time variant data

# Generated columns are computed based on row data

```
CREATE TABLE table_name (
    Col1 DATATYPE,
    Col2 DATETYPE,
    GeneratedCol DATATYPE [GENERATED ALWAYS] AS (expression) [VIRTUAL | STORED]
);
```

"GENERATED ALWAYS" optional

VIRTUAL: computed on demand

STORED: Stored on disk

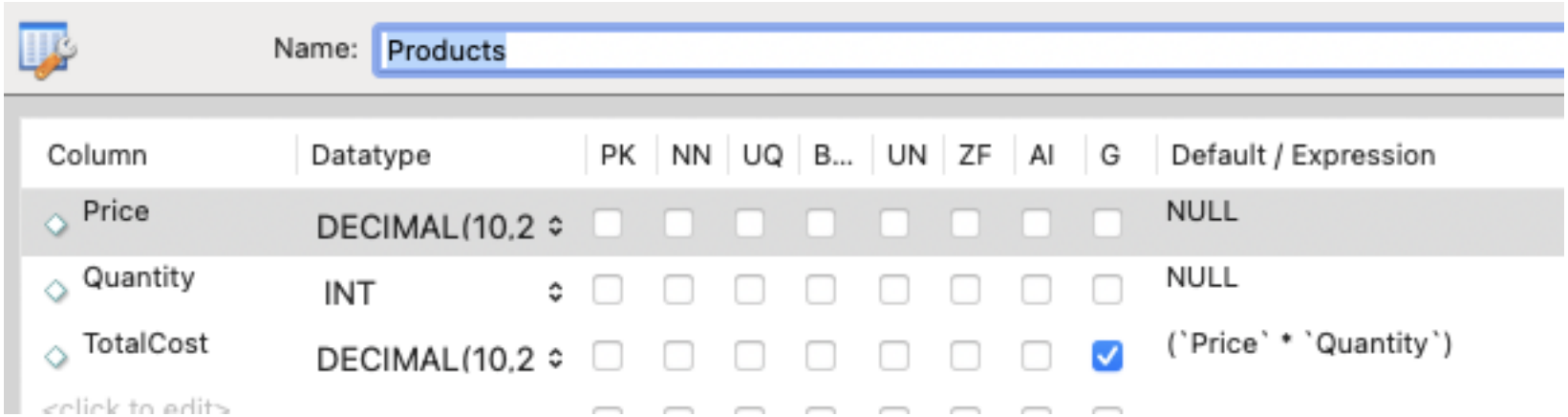
Cannot reference other tables in expression, can only reference attributes in this table

| | Virtual | Stored |
|---|---|---|
| | Values computed on the fly, NOT stored on disk | Values stored on disk Compute when a row is inserted or updated |
| Disk storage | None | Yes |
| Read speed | Slower | Faster |
| Write speed | Faster | Slower |
| Use case | Infrequent reads, saves space | Frequent reads, complex expressions |

# Use case: Calculate a generated total value

```
CREATE TABLE Products (
    Price DECIMAL(10,2),
    Quantity INT,
    TotalCost DECIMAL(10,2) GENERATED ALWAYS AS (Price * Quantity) VIRTUAL
);
```

TotalCost computed on demand



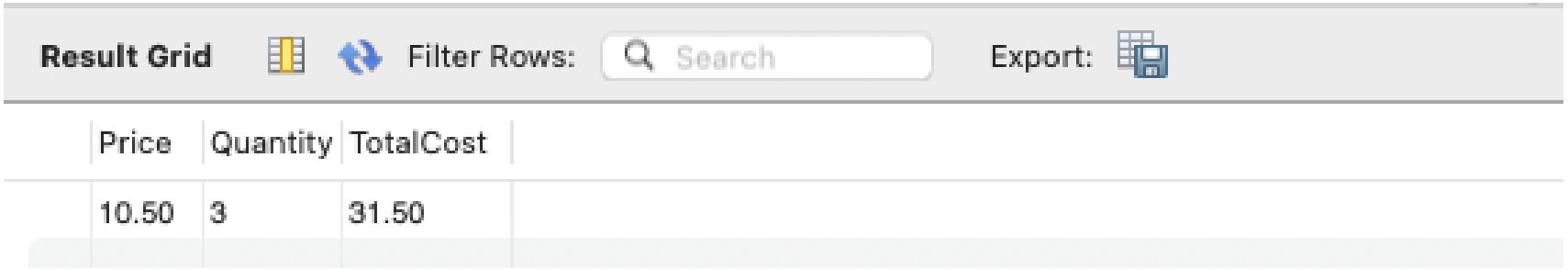
# Use case: Calculate a generated total value

```
CREATE TABLE Products (
    Price DECIMAL(10,2),
    Quantity INT,
    TotalCost DECIMAL(10,2) GENERATED ALWAYS AS (Price * Quantity) VIRTUAL
);

INSERT INTO Products (Price, Quantity) VALUES (10.50, 3);

SELECT * FROM Products;
```

TotalCost computed on demand



# Use case: Add a generated InspectionYear column to restaurant_inspections

```
USE nyc_data;
ALTER TABLE restaurant_inspections
    ADD COLUMN InspectionYear INT
        GENERATED ALWAYS AS (YEAR(InspectionDate)) VIRTUAL;

SELECT *
    FROM restaurant_inspections
    WHERE InspectionDate <> '1900-01-01';
```

**YEAR is a DateTime function to extract the year from a DATE**

**InspectionYear holds the year calculated from InspectionDate**

**Now can query for Inspections in a particular year**

Result Grid | Filter Rows: Search | Export: | Fetch rows:

| | CRITICAL FLAG | SCORE | GRADE | GRADE DATE | RECORD DATE | INSPECTION TYPE | InspectionDa... | GradeDate | RecordDate | InspectionYear |
|---|---|---|---|---|---|---|---|---|---|---|
| h facility's food and/or non-f... | Critical | NULL | | | 12/02/2025 | Cycle Inspection / Initial Inspection | 2022-06-22 | NULL | 2025-12-02 | 2022 |
| ing and sanitizing of tablew... | Not Critical | 51 | C | 05/22/2024 | 12/02/2025 | Cycle Inspection / Re-inspection | 2024-05-22 | 2024-05-22 | 2025-12-02 | 2024 |
| or in a sanitizing solution,... | Critical | 16 | | | 12/02/2025 | Cycle Inspection / Initial Inspection | 2025-03-04 | NULL | 2025-12-02 | 2025 |
| | Not Applicable | NULL | P | 03/03/2023 | 12/02/2025 | Cycle Inspection / Reopening Inspection | 2023-03-03 | 2023-03-03 | 2025-12-02 | 2023 |
| or conditions conducive to r... | Not Critical | 26 | | | 12/02/2025 | Cycle Inspection / Initial Inspection | 2023-06-13 | NULL | 2025-12-02 | 2023 |
| | Not Critical | 8 | A | 09/12/2025 | 12/02/2025 | Cycle Inspection / Initial Inspection | 2025-09-12 | 2025-09-12 | 2025-12-02 | 2025 |
| 140 °F. | Critical | 25 | | | 12/02/2025 | Cycle Inspection / Initial Inspection | 2024-04-30 | NULL | 2025-12-02 | 2024 |
| | Not Critical | 13 | A | 05/09/2025 | 12/02/2025 | Cycle Inspection / Re-inspection | 2025-05-09 | 2025-05-09 | 2025-12-02 | 2025 |
| od area. | Critical | 15 | B | 08/31/2023 | 12/02/2025 | Cycle Inspection / Re-inspection | 2023-08-31 | 2023-08-31 | 2025-12-02 | 2023 |

# Use case: Extract JSON fields from DartAir

```
CREATE TABLE Employees3 (
    EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
    FullName VARCHAR(100) NOT NULL,
    Salary DECIMAL(10, 2),
    HireDate DATE,
    EmployeeType ENUM('Pilot', 'Mechanic', 'Accountant', 'Other') NOT NULL,
    Properties JSON DEFAULT NULL,
    FlightHours INT AS (JSON_UNQUOTE(Properties->'$.FlightHours')) STORED
);
```

**Note: "GENERATED" is optional, but recommended!**

**Flight hours returned for pilots, others are NULL**

Result Grid | Filter Rows: Search | Edit: | Export/Import:

| EmployeeID | FullName | Salary | HireDate | EmployeeType | Properties | FlightHours |
|---|---|---|---|---|---|---|
| 1 | Ace Pilot | 110000.00 | 2026-04-11 | Pilot | {"FlightHours": 2222, "LicenseType": "ATP", "Me… | 2222 |
| 2 | Joe Mechanic | 77000.00 | 2026-04-02 | Mechanic | {"CertificationType": "A&P", "LastSafetyTraining"… | NULL |
| 3 | Alice Accountant | 123000.00 | 2026-04-05 | Accountant | {"Title": "CPA Accountant", "CPADate": "2026-4-5"} | NULL |
| NULL | NULL | NULL | NULL | NULL | NULL | NULL |

# Agenda

1. Views
2. Object-oriented data
3. Generated columns
4. Time variant data

# Updates overwrite data, but sometimes we want to know how things change over time

```
296 •  SELECT e.*, p.*
297        FROM Pilots p LEFT JOIN Employees e ON p.EmployeeID = e.EmployeeID;
298
```

100% 14:297

Result Grid Filter Rows: Search Export:

| EmployeeID | FullName | Salary | HireDate | EmployeeType | EmployeeID | LicenseType | FlightHours | MedicalType |
|---|---|---|---|---|---|---|---|---|
| 1 | Chuck Yeager | 100000.00 | 1960-06-01 | Pilot | 1 | ATP | 5000 | III |

**Chuck Yeager makes $100,000**

```
299    -- Salary history lost if update Chuck's salary
300 •  UPDATE Employees
301        SET Salary = 120000 WHERE EmployeeID = 1;
302 •  SELECT e.*, p.*
303        FROM Pilots p LEFT JOIN Employees e ON p.EmployeeID = e.EmployeeID;
304
```

**Updating salary looses old salary**
**Sometimes we want to know how things change over time**

100% 10:302

Result Grid Filter Rows: Search Export:

| EmployeeID | FullName | Salary | HireDate | EmployeeType | EmployeeID | LicenseType | FlightHours | MedicalType |
|---|---|---|---|---|---|---|---|---|
| 1 | Chuck Yeager | 120000.00 | 1960-06-01 | Pilot | 1 | ATP | 5000 | III |

**See current salary, but have no idea of previous salary**

# We can use a history table to see how data changes over time

```
CREATE TABLE EmployeeSalaryHistory(
    EmployeeID INT NOT NULL,
    Salary DECIMAL(10, 2),
    FromDate DATE,
    ToDate DATE DEFAULT NULL,
    PRIMARY KEY (EmployeeID, FromDate),
    CONSTRAINT fk_salary_history_employee
        FOREIGN KEY (EmployeeID) REFERENCES Employees(EmployeeID)
            ON DELETE RESTRICT
            ON UPDATE CASCADE
);
```

EmployeeID is a foreign key on Employees

New salary

FromDate is effective date, ToDate NULL

**When a new employee is hired:**

- **Insert an entry with their starting salary**
- **FromDate = HireDate**
- **Leave ToDate NULL**

| EmployeeID | Salary | FromDate | ToDate |
|---|---|---|---|
| 1 | 120000.00 | 1960-06-01 | NULL |
| 2 | 70000.00 | 1992-07-15 | NULL |
| 3 | 350000.00 | 2023-05-17 | NULL |
| 4 | 45000.00 | 2026-02-27 | NULL |
| NULL | NULL | NULL | NULL |

# Updating salary requires three queries

```
-- give Chuck Yeager a raise
SET @effectiveDate = '2026-4-16';
SET @employeeID = 1;
SET @newSalary = 125000;
-- update history, finish old salary
UPDATE EmployeeSalaryHistory
    SET ToDate = @effectiveDate
    WHERE EmployeeID = @employeeID
        AND ToDate IS NULL;
-- insert entry for new salary
INSERT INTO EmployeeSalaryHistory
    VALUES (@employeeID, @newSalary, @effectiveDate, NULL);

-- update employees
UPDATE Employees
    SET Salary = @newSalary
    WHERE EmployeeID = @employeeID;
```

**Set variables for convenience**

**Set ToDate to finish old Salary for Chuck in *EmployeeSalaryHistory* table**

**Make new entry with FromDate set to the effective date of salary and NULL ToDate**

**Update *Employees* table with current salary**

# We can use a history table to see how data changes over time

```
-- give Chuck Yeager a raise
SET @effectiveDate = '2026-4-16';
SET @employeeID = 1;
SET @newSalary = 125000;
-- update history, finish old salary
UPDATE EmployeeSalaryHistory
    SET ToDate = @effectiveDate
    WHERE EmployeeID = @employeeID
        AND ToDate IS NULL;
-- insert entry for new salary
INSERT INTO EmployeeSalaryHistory
    VALUES (@employeeID, @newSalary, @effectiveDate, NULL);

-- update employees
UPDATE Employees
    SET Salary = @newSalary
    WHERE EmployeeID = @employeeID;
```

**Employees**

| EmployeeID | FullName | Salary | HireDate | EmployeeType |
|---|---|---|---|---|
| 1 | Chuck Yeager | 125000.00 | 1960-06-01 | Pilot |
| 2 | Snuffy Smith | 70000.00 | 1992-07-15 | Mechanic |
| 3 | Scrooge McDuck | 350000.00 | 2023-05-17 | Accountant |
| 4 | Nobody Special | 45000.00 | 2026-02-27 | Other |

**EmployeeSalaryHistory**

| EmployeeID | Salary | FromDate | ToDate |
|---|---|---|---|
| 1 | 125000.00 | 2026-04-16 | NULL |
| 1 | 120000.00 | 1960-06-01 | 2026-04-16 |
| 2 | 70000.00 | 1992-07-15 | NULL |
| 3 | 350000.00 | 2023-05-17 | NULL |
| 4 | 45000.00 | 2026-02-27 | NULL |

**Chuck has $125K salary as of 2026-04-16!**
**From 1960 to 2026 his salary was $120K**

**Preview: what happens if all three queries do not complete?**
**Transaction can keep data consistent!**

# Practice

Use nyc_data

Create a Grade History table for Restaurants

- Make RestaurantID a foreign key referencing Restaurants
- Ensure that Restaurants cannot be deleted if there are entries in the history table for the restaurant to be deleted, cascade updates
- Fill the history table with the most recent grade for each restaurant (hint: use a correlated subquery)
- When a new inspection grade is posted for an inspection (e.g., where Grade <> ‘’)
  - Close the prior grade date range in the history table
  - Make a new entry in the history table starting with the inspection date and the current grade (leave ending date NULL)
- Test that your solution works