How many distinct last names of actors are there?

```
SELECT COUNT(DISTINCT last_name)
FROM actor;//
```

Which actors participated in the movie 'Academy Dinosaur'? Print their first and last names.

```
SELECT first_name, last_name
FROM actor AS a
JOIN film_actor AS fa
ON a.actor_id = fa.actor_id
JOIN film AS f
ON f.film_id = fa.film_id
WHERE f.title = 'Academy Dinosaur';
```

3. How many copies of the film 'Hunchback Impossible' exist in the inventory system?

```
SELECT COUNT(*) AS copies_in_inventory
FROM inventory AS i
JOIN film AS f
ON i.film_id = f.film_id
WHERE f.title = 'Hunchback Impossible';
```

4. What is the total amount paid by each customer for all their rentals? For each customer print their name and the total amount paid.

```
SELECT c.first_name, c.last_name, SUM(p.amount) AS total_amount
FROM customer AS c
JOIN payment AS p
ON c.customer_id = p.customer_id
GROUP BY c.customer_id;
```

5. How many films from each category each store has? Print the store id, category name and number of films. Order the results by store id and category name.

```
SELECT i.store_id, c.name AS category, COUNT(*) AS films_num
FROM category AS c
JOIN film_category AS fc
ON c.category_id = fc.category_id
JOIN inventory AS i
ON i.film_id = fc.film_id
GROUP BY i.store_id, c.category_id
ORDER BY i.store_id, c.name;
```

6. Calculate the total revenue of each store.

```
SELECT i.store_id, SUM(amount) AS total_revenue
FROM payment AS p
JOIN rental AS r
ON p.rental_id = r.rental_id
JOIN inventory AS i
ON i.inventory_id = r.inventory_id
GROUP BY i.store_id;
```

7. Which actor participated in the most films? Print their full name and in how many movies they participated

```
SELECT first_name, last_name, COUNT(*) AS films_num
FROM actors AS a
JOIN film_actor AS fa
ON a.actor_id = fa.actor_id
GROUP BY a.actor_id
HAVING COUNT(*) >= ALL (
  SELECT COUNT(*)
  FROM film_actor
  GROUP BY actor_id);
```

8. Find pairs of actors that participated together in the same movie and print their full names. Each such pair should appear only once in the result.

```
SELECT DISTINCT CONCAT(a1.first_name, ' ', a1.last_name) AS first_actor,
CONCAT(a2.first_name, ' ', a2.last_name) AS second_actor
FROM actor AS a1
JOIN film_actor AS fa1
ON a1.actor_id = fa1.actor_id
JOIN film_actor AS fa2
ON fa1.film_id = fa2.film_id
JOIN actor AS a2
ON a2.actor_id = fa2.actor_id
WHERE a1.actor_id < a2.actor_id;
```

9. Display the top five most popular films, i.e., films that were rented the highest number of times. For each film print its title and the number of times it was rented

```
SELECT title, COUNT(*) AS times_rented
FROM film AS f
JOIN inventory AS i
ON f.film_id = i.film_id
JOIN rental AS r
ON i.inventory_id = r.inventory_id
GROUP BY f.film_id
ORDER BY times_rented DESC
LIMIT 5;
```

10. Is the film 'Academy Dinosaur' available for rent from Store 1? You should check that the film exists as one of the items in the inventory of Store 1, and that there is no outstanding rental of that item with no return date.

```
SELECT EXISTS (
  SELECT *
  FROM inventory AS i
  JOIN film AS f
  ON i.film_id = f.film_id
  WHERE f.title = 'Academy Dinosaur'
    AND i.store_id = 1
    AND NOT EXISTS (
      SELECT *
      FROM rental AS r
      WHERE i.inventory_id = r.inventory_id
        AND r.return_date IS NULL) ) AS 'available';
```

For each department, find the maximum salary of instructors in that department.

```
SELECT dept_name, MAX(salary) AS max_salary
FROM instructor
GROUP BY dept_name;
```

Find the IDs and names of those instructors who have taught every course in their department.

```
SELECT i.ID, i.name
FROM instructor i
JOIN teaches t ON i.ID = t.ID
JOIN course c ON t.course_id = c.course_id
GROUP BY i.ID, i.name, c.dept_name
HAVING COUNT(DISTINCT t.course_id) = (SELECT COUNT(DISTINCT course_id)
FROM course WHERE dept_name = c.dept_name);
```

Find the IDs of those students who have retaken at least three distinct courses at least once (i.e., the student has taken the course at least two times).

```
SELECT t.ID
FROM takes AS t
GROUP BY t.ID
HAVING COUNT(DISTINCT t.course_id) >= 3
AND COUNT(CASE WHEN t.course_id IS NOT NULL THEN t.course_id END) >= 6;
```

13. For each student who have retaken a course at least once (i.e., the student has taken the course at least twice), show the student's ID, name and the course ID.

```
SELECT s.ID, s.name, t.course_id
FROM student AS s
JOIN takes AS t ON s.ID = t.ID
GROUP BY s.ID, s.name, t.course_id
HAVING COUNT(t.course_id) >= 2;
```

12. Rewrite the preceding query, but also ensure that you include only instructors who have given at least one other non-null grade in some course.

```
SELECT DISTINCT i.ID, i.name
FROM instructor AS i
LEFT JOIN teaches AS t ON i.ID = t.ID
LEFT JOIN takes AS tk ON t.course_id = tk.course_id AND t.sec_id = tk.sec_id
WHERE i.ID NOT IN (
   SELECT t.ID
   FROM teaches AS t
   JOIN takes AS tk ON t.course_id = tk.course_id AND t.sec_id = tk.sec_id
   WHERE tk.grade IN ('A', 'A-') )
AND i.ID IN (
   SELECT t.ID
   FROM teaches AS t
   JOIN takes AS tk ON t.course_id = tk.course_id AND t.sec_id = tk.sec_id
   WHERE tk.grade IS NOT NULL);
```

Find the ID and name of each instructor who has never given an 'A' grade in any course she or he has taught. (Instructors who have never taught a course trivially satisfy this condition.)

```
SELECT DISTINCT i.ID, i.name
FROM instructor AS i
LEFT JOIN teaches AS t ON i.ID = t.ID
LEFT JOIN takes AS tk ON t.course_id = tk.course_id AND t.sec_id = tk.sec_id
WHERE i.ID NOT IN (
   SELECT t.ID
   FROM teaches AS t
   JOIN takes AS tk ON t.course_id = tk.course_id AND t.sec_id = tk.sec_id
   WHERE tk.grade = 'A');
```

Find the ID and name of each History student who has not taken any Music courses.

```
SELECT s.ID, s.name
FROM student AS s
WHERE s.dept_name = 'History'
AND NOT EXISTS (
   SELECT 1
   FROM takes AS t
   JOIN course AS c ON t.course_id = c.course_id
   WHERE s.ID = t.ID
   AND c.dept_name = 'Music');
```

Find the ID and name of the student with the highest number of 'A' grades (there may be more than one such student)

```
SELECT s.ID, s.name, COUNT(t.grade) AS a_grade_count
FROM student AS s
JOIN takes AS t ON s.ID = t.ID
WHERE t.grade = 'A'
GROUP BY s.ID, s.name
HAVING COUNT(t.grade) = (
   SELECT MAX(a_grade_count)
   FROM (
      SELECT COUNT(grade) AS a_grade_count
      FROM takes
      WHERE grade = 'A'
      GROUP BY ID
   ) AS subquery);
```

Find the enrollment (number of students) in each section that was offered in Spring 2017. The result columns should be course id, section id, students num. You do not need to output sections with 0 students.

```
SELECT sec.course_id, sec.sec_id, COUNT(*) AS student_num
FROM takes AS t
INNER JOIN section AS sec
ON t.course_id = sec.course_id
AND t.sec_id = sec.sec_id
AND t.semester = sec.semester
AND t.year = sec.year
WHERE sec.year = 2017 AND sec.semester = 'Spring'
GROUP BY sec.course_id, sec.sec_id
HAVING COUNT(*) > 0;
```

Find the IDs and names of all instructors earning the highest salary (there may be more than one with the same salary).

```
SELECT ID, name
FROM instructor
WHERE salary = (SELECT MAX(salary) FROM instructor);
```

Find the IDs and titles of all the courses that are (direct) prerequisite to the Robotics course.

```
SELECT c1.course_id AS prerequisite_id, c1.title AS prerequisite_title
FROM course AS c
JOIN prereq AS p
ON p.course_id = c.course_id
JOIN course AS c1
ON p.prereq_id = c1.course_id
WHERE c.title = 'Robotics';
```

For each English letter show how many student names start with that letter

```
SELECT LEFT(name, 1) as first_let, count(*) as Student_num
FROM student
Group by LEFT(name,1)
ORDER BY first_let
```

Find all the instructors whose salary is higher than Katz's salary

```
Select i1.name, i1.salary
From instructor i1
Join instructor i2
On i1.salary> i2.salary
Where i2.name ='Katz';
```

Find all instructors whose salary is higher than the average salary in their own department

```
SELECT i1.name, i1.salary
FROM instructor i1
WHERE salary > (
   SELECT AVG(salary)
   FROM instructor i2
   WHERE i2.dept_name = i1.dept_name);
```

Find the department with the highest number of courses

```
SELECT dept_name, COUNT(*) AS courses_num
FROM course
GROUP BY dept_name
HAVING COUNT(*) >= ALL (
   SELECT COUNT(*)
   FROM course
   GROUP BY dept_name)
```

Find all the courses that were offered both in Fall 2017 and Spring 2018"

```
SELECT DISTINCT course_id
FROM section AS s1
WHERE semester = 'Fall' AND year = 2017
AND EXISTS (
   SELECT *
   FROM section AS s2
   WHERE semester = 'Spring' AND year = 2018
   AND s1.course_id = s2.course_id)
```

Find the highest number of courses offered in any given department

```
SELECT MAX(courses_in_dept.courses_num)
FROM (
   SELECT COUNT(*) courses_num
   FROM course
   GROUP BY dept_name
) AS courses_in_dept;
```

```
DELIMITER //

CREATE PROCEDURE enroll_student_in_section (
   IN input_student_id VARCHAR(5),
   IN input_course_id VARCHAR(8),
   IN input_sec_id VARCHAR(8),
   IN input_semester VARCHAR(6),
   IN input_year DECIMAL(4,0)
)
BEGIN
   DECLARE prereq_count INT;
   DECLARE completed_count INT;
   DECLARE prereq_missing INT;
   DECLARE already_enrolled INT;

   SELECT COUNT(*) INTO already_enrolled
   FROM takes
   WHERE ID = input_student_id
     AND course_id = input_course_id
     AND sec_id = input_sec_id
     AND semester = input_semester
     AND year = input_year;

   IF already_enrolled > 0 THEN
      SELECT CONCAT('Student ', input_student_id, ' is already enrolled in course ',
input_course_id, ' section ', input_sec_id) AS result;
   ELSE

      SELECT COUNT(*) INTO prereq_count
      FROM prereq
      WHERE course_id = input_course_id;

      SELECT COUNT(*) INTO completed_count
      FROM prereq AS p
      JOIN takes AS t ON p.prereq_id = t.course_id
      WHERE t.ID = input_student_id
        AND t.grade IS NOT NULL
        AND t.grade <> 'F'
        AND p.course_id = input_course_id;

      IF completed_count = prereq_count THEN

         INSERT INTO takes (ID, course_id, sec_id, semester, year, grade)
         VALUES (input_student_id, input_course_id, input_sec_id, input_semester,
input_year, NULL);

         SELECT CONCAT('Student ', input_student_id, ' successfully enrolled in
course ', input_course_id, ' section ', input_sec_id) AS result;
      ELSE

         SET prereq_missing = prereq_count - completed_count;
         SELECT CONCAT('Student ', input_student_id, ' cannot be enrolled in course
', input_course_id, ' due to missing ', prereq_missing, ' prerequisite(s).') AS result;
      END IF;
   END IF;

END //

DELIMITER ;
Call enroll_student(`'`,`'`,`'`,)
```

DML

Create a new course "CS-001" in the Comp. Sci. department titled "Weekly Seminar," with 2 credits.

```
INSERT INTO course (course_id, title, dept_name, credits)
VALUES ('CS-001', 'Weekly Seminar', 'Comp. Sci.', 2);
```

Create a section of this course in Spring 2022, with sec id of 1, and with the location of this section not yet specified.

```
INSERT INTO section (course_id, sec_id, semester, year, building, room_number)
VALUES ('CS-001', 1, 'Spring', 2022, NULL, NULL);
```

Enroll every student in the Comp. Sci. department in the above section.

```
INSERT INTO takes (ID, course_id, sec_id, semester, year, grade)
SELECT s.ID, 'CS-001', 1, 'Spring', 2022, NULL
FROM student AS s
WHERE s.dept_name = 'Comp. Sci.';
```

Delete enrollments in the above section where the student's ID is 12345.

```
DELETE FROM takes
WHERE course_id = 'CS-001' AND sec_id = 1 AND semester = 'Spring' AND year =
2022 AND ID = '12345';
```

Delete the course "CS-001." What happened to the section and enrollments of this course?

```
DELETE FROM course
WHERE course_id = 'CS-001';
```

DDL

```
CREATE TABLE person (
   driver_id INT PRIMARY KEY,
   name VARCHAR(50) NOT NULL,
   address VARCHAR(150)          );
 CREATE TABLE car (
   license_plate VARCHAR(15) PRIMARY KEY,
   model VARCHAR(50) NOT NULL,
   year INT CHECK (year >= 1886)          );
CREATE TABLE owns (
   driver_id INT,
   license_plate VARCHAR(15),
   PRIMARY KEY (driver_id, license_plate),
   FOREIGN KEY (driver_id) REFERENCES person(driver_id) ON DELETE CASCADE,
   FOREIGN KEY (license_plate) REFERENCES car(license_plate) ON DELETE
CASCADE );
 CREATE TABLE accident (
   report_number INT PRIMARY KEY,
   date DATE NOT NULL,
   location VARCHAR(150) NOT NULL     );
CREATE TABLE participated (
   report_number INT,
   license_plate VARCHAR(15),
   driver_id INT,
   damage_amount DECIMAL(10, 2),
   PRIMARY KEY (report_number, license_plate, driver_id),
   FOREIGN KEY (report_number) REFERENCES accident(report_number) ON
DELETE CASCADE,
   FOREIGN KEY (license_plate) REFERENCES car(license_plate) ON DELETE
CASCADE,
   FOREIGN KEY (driver_id) REFERENCES person(driver_id) ON DELETE CASCADE);
```

Find all the clients that have been trained in a coaching type whose description contains the word "life".

```
SELECT tp.client_id
FROM training_program AS tp
JOIN types AS t
ON tp.type_name = t.type_name WHERE t.description LIKE '%life%';
```

Find pairs of different clients who started their training at the same day with the same coach. The result columns should include the ids of the customers and the starting date of the training. Each such pair should appear only once in the result.

```
SELECT t1.client_id, t2.client_id, t1.start_date
FROM training_program AS t1
JOIN training_program AS t2
ON t1.client_id < t2.client_id
WHERE t1.start_date = t2.start_date AND t1.coach-id = t2.coach-id;
```

Find all the clients who have never been trained by a coach named Levi.

```
SELECT client_id
 FROM clients
 WHERE client_id NOT IN (
SELECT tp.client_id
FROM training_program AS tp JOIN coach AS c
ON tp.coach_id = c.coach_id WHERE c.name = 'Levi' );
```

Find for each client the total amount he/she has to pay for all his/her training programs.

```
SELECT tp.client_id, SUM(tp.hours * c.hourly_rate) AS total_pay
 FROM training_program AS tp
 JOIN coach AS c
 ON tp.coach_id = c.coach_id
GROUP BY tp.client_id;
```

Find customers who have been trained in all the coaching types offered by the club.

```
SELECT c.client_id
 FROM clients AS c
 WHERE NOT EXISTS (
    SELECT *
    FROM types
    WHERE type_name NOT IN (
       SELECT tp.type_name
       FROM training_program AS tp
       WHERE tp.client_id = c.client_id
) );
```

Find customers who have had at least 3 different training programs, and all their training programs have been carried out by the same coach.

```
SELECT client_id
 FROM training_program
 WHERE client_id NOT IN (
    SELECT t1.client_id
    FROM training_program AS t1
    JOIN training_program AS t2
    ON t1.client_id = t2.client_id
    WHERE t1.coach_id <> t2.coach_id
 )
 GROUP BY client_id
 HAVING COUNT(*) >= 3;
```

Find how many flights departed from LAX airport in July 2015.
```
flights[(flights['ORIGIN_AIRPORT'] == 'LAX') & (flights['YEAR'] == 2015) &
(flights['MONTH'] == 7)
```
Find the number of the flight that had the longest arrival delay
```
flights.loc[flights['ARRIVAL_DELAY'].idxmax(),'FLIGHT_NUMBER']
```
Find the airport with the highest number of arriving flights.
```
flights.groupby('DESTINATION_AIRPORT').size().idxmax()
```
Find the day of week that had the highest number of flight cancellations.
```
flights.groupby('DAY_OF_WEEK')['CANCELLED'].sum().idxmax()
```
Create a bar plot showing the number of flights for each airline.
```
flights.groupby('AIRLINE').size().plot.bar()
```
Find the mean arrival delay for each airline.
```
flights.groupby('AIRLINE')['ARRIVAL_DELAY'].mean()
```
Find the airlines that had more than 10,000 cancellations.
```
df = flights[flights['CANCELLED'] == 1].groupby('AIRLINE').size()
df[df > 10000]
```
Find airlines having more than 2% of their flights cancelled. For each such airline, print its identifier and the percentage of cancelled flights.
```
df = flights.groupby('AIRLINE')['CANCELLED'].mean()
df[df > 0.02]
```
Find the three top airlines with the highest number of cancelled or diverted flights.
```
flights[(flights['CANCELLED'] == 1) |
(flights['DIVERTED'] == 1)]['AIRLINE'].value_counts()[:3]
```
Find the longest sequence of on-time flights for each airline (an on-time flight is a flight with less than 15 minutes arrival delay).
```
df['ontime'] = df['ARRIVAL_DELAY'] < 15
df.groupby('AIRLINE')['ontime'].apply(lambda x:
(~x).cumsum().where(x).value_counts().max())
```
Find the three airlines with the longest arrival delays.
```
Flights.groupby('Airline')['Arrival_Delay'].max().nlargest(3)
```
name and age of players who play for 'Boston Eagles' and their age is above 28.
```
df[(df['Team'] == 'Boston Eagles') & (df['Age'] > 28)][['Name', 'Age']]
```
the name and height of the tallest player.
```
df.loc[df['Height'].idxmax(), ['Name', 'Height']]
```
players whose salary is above the average salary in their team
```
df['Avg_Team_Salary'] = df.groupby(['Team'])['Salary'].transform('mean')
df[df['Salary'] >
df['Avg_Team_Salary']][['Name','Team','Salary','Avg_Team_Salary']]
```

Given an array A, print all the numbers that appear more than twice in the array.
```
values, counts = np.unique(A, return_counts=True)
values[counts > 2]
```
Given an array A and a number v, find the closest value in A to v
```
A[np.argmin(np.abs(A - v))]
```
Given a matrix A, print the number of rows in the matrix that contain the number 5
```
np.count_nonzero(np.any(A == 5, axis=1))
```
Given a matrix A, print the indices of the columns that contain a number greater than 7
```
np.where(np.any(A > 7, axis=0))[0]
```
Given a matrix A, zero out all the rows that contain a negative number
```
A[np.where(np.any(A < 0, axis=1))] = np.zeros(A.shape[1])
```
Given a matrix A, normalize its rows such that the numbers in each row sum to 1.
```
A = A / np.sum(A, axis=1).reshape(-1, 1)
np.apply_along_axis(lambda row: row/np.sum(row) if np.sum(row) != 0 else row,
axis =1, arr = matrix)
```
Given an array A, find all the even numbers in the array that are followed by an odd number.
```
B = A[:-1]
C = A[1:]
B[(B % 2 == 0) & (C % 2 == 1)]
```
Given an array A, find all the local maxima (peaks) in the array. A local maximum is a number that is greater or equal to its two neighbors
```
B = A[1:-1]  # the first and last numbers cannot be peaks
B[(B > A[:-2]) & (B > A[2:])]
```
Given a matrix A, find how many rows in A contain duplicate values.
```
B = np.sort(A, axis=1)
np.count_nonzero(np.any(B[:, 1:] == B[:, :-1], axis=1))
```
*Given a matrix A, find all the **saddle points** in A.*
```
N = np.argmin(A, axis=1)# change min -> max after
M = np.argmax(A, axis=0) # same
saddle_row = np.where(M[N] == np.arange(A.shape[0]))[0]
A[saddle_row, N[saddle_row]]
```
#latin square
```
def is_latin_square(matrix):
    n = matrix.shape[0]
    seq = np.arange(1, n + 1)
    rows_check = np.all(np.sort(matrix, axis=0) == seq.reshape(-1, 1))
    cols_check = np.all(np.sort(matrix, axis=1) == seq)
    return rows_check and cols_check
minimum = q1.min()
maximum = q1.max()
```
The average of every even row
```
answer_b = q1[::2].mean(axis=1)
```
The standard of deviation in every odd columnn
```
answer_c = q1[:,1::2].std(axis=0)
```
How many rows contain a number less than 5?
```
row_less5 = np.sum(np.any(q1<5, axis=1))
```
Which column contain a number greater than 90? (
```
np.where(np.any(q1 > 90, axis = 0))[0]
```
which numbers appear in the matrix more than twice?
```
values, counts = np.unique(q1, return_counts = True)
num_2 = values[counts > 2]
```
How many row contains duplicate value?
```
row_duplicate = np.sum(np.apply_along_axis(lambda x :len(np.unique(x))!=
len(x), axis = 1, arr = q1 ))
#matplotlib
years = [1972, 1974, 1978, 1982, 1985, 1989, 1993, 1997, 1999, 2000, 2003,
2004, 2007, 2008, 2012]
transistors=[0.0024, 0.005, 0.029, 0.12, 0.275, 1.18, 3.1, 7.5, 24.0, 42.0, 220.0,
592.0,1720.0, 2046.0, 3100.0]
y0 = 1972
n0 = transistors[0]
expected_transistors = [n0 * 2**((year- y0)/2) for year in years]
log_actual = np.log(transistors)
log_expected = np.log(expected_transistors)
plt.figure(figsize = (8,4))
plt.plot(years, log_expected,'--',label = "Moore's laws prediction")
plt.scatter(years, log_actual, marker ='*',color='r', label = "Actual Transistor")
plt.xlabel("year")
plt.ylabel("Transistors(in millions) in log scale")
plt.legend()
plt.title("Moore's law vs actual transistors")
plt.grid(True)
```

```
browsers = ['Chrome', 'Safari', 'Edge', 'Firefox', 'Samsung Internet', 'Opera']
market_share = [65.18, 18.55, 5.26, 2.74, 2.56, 2.15]
colors = ['gold','silver','yellowgreen', 'lightcoral', 'lightskyblue','pink']
explode = (0.1, 0, 0, 0, 0, 0)
plt.title('Browser market shares worldwide(2024)')
plt.pie(market_share, explode = explode, labels= browsers, colors= colors, autopct
= '%1.1f%%', shadow= True, startangle=60, labeldistance=1.15)
plt.axis('equal')
```

```
plt.figure(figsize = (8,4))
plt.scatter(sepal_length, sepal_width, s = petal_length *50, c = y, cmap = 'viridis',
edgecolor = 'k', alpha = 0.5)
plt.xlabel('sepal length(cm)')
plt.ylabel('sepal width(cm)')
plt.colorbar()
plt.grid(True)
plt.title("The Iris Flower Data Set")
```

```
fig, axes = plt.subplots(5, 10, figsize = (10, 5))
axes = axes.ravel()
for i in range(50):
    axes[i].imshow(images[i], cmap='gray')
    axes[i].set_title(label[i])
    axes[i].axis('off')
    plt.tight_layout()
```

```
from mpl_toolkits.mplot3d import Axes3D
x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(X**2 + Y**2)
fig = plt.figure()
ax = plt.axes(projection = '3d')
surface = ax.plot_surface(X, Y, Z, cmap='coolwarm')
fig.colorbar(surface)
plt.show()
```

DBMS Architecture/ A DBMS consists of two major components
-Storage manager -Query processor
Good Database Design Principles/ Avoid duplicate information (redundancy)
Store information in its smallest logical parts Define a primary key for each table
If there is no column that might make a good primary key, use an auto-increment column Ensure the integrity of the data by applying integrity constraints Apply normalization rules
DDL(Data DefinitionLanguage) CREATE ALTER DROP RENAME TRUNCATE
DML(Data ManipulationLanguage)INSERT UPDATE DELETE MERGE CALL EXPLAIN PLAN LOCK TABLE DQL(Data QueryLanguage)SELECT DCL(Data ControlLanguage) GRANT REVOKE TCL(Transaction ControlLanguage) COMMIT ROLLBACK SAVEPOINT SET TRANSACTION
Primary Key
The primary key uniquely identifies each record in the table
☐ A table can have only one primary key
☐ The primary key can consist of one or more columns
☐ The columns of the primary key are automatically defined as unique and not null
☐ The primary key should be chosen such that its values are never/rarely changed
☐ The primary key can be defined as part of a column declaration or appear on its own
☐ A foreign key is a column (or set of columns) in one table that refers to a primary key
in another table
☐ The table with the foreign key is called the child table
☐ The table with the primary key is called the parent table
☐ Foreign keys enforce referential integrity of the data
☐ A value in the foreign key column must be one of the values contained in the primary key
☐ ON DELETE CASCADE – when a row in the parent table is deleted, automatically delete its
corresponding child rows
☐ ON UPDATE CASCADE – when the primary key column in the parent table is updated,
automatically update the foreign key column in the corresponding child rows
☐ ALTER TABLE allows you change the table definition a=ALTER TABLE table_name
☐ To add a column to a table, use the following:\ a ADD column_name datatype;
☐ To delete a column from a table: a DROP COLUMN column_name;
☐ change the data type of a column: a MODIFY COLUMN column_name datatype;
☐ The DROP TABLE statement allows you to drop an existing table
DROP TABLE table_name; - Deletes both the data and the schema of the table
☐ TRUNCATE TABLE deletes only the data in the table, but not the table itself
TRUNCATE TABLE table_name;
☐ It's faster than DELETE if you want to delete all the records from the table
☐ **SQL DML** commands allows you to manipulate the data in the database
☐ Insertion of new rows into a given table
☐ Deletion of rows in a given table☐ Updating values in a given table
UPDATE employees
SET salary = salary * 1.1
WHERE employee_id = 1;

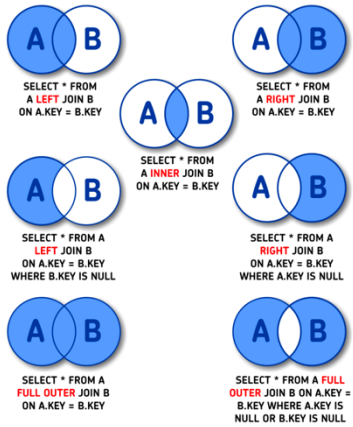
DELETE FROM employees
WHERE employee_id = 1;

Select
CASE
  WHEN condition1 THEN result1
  WHEN condition2 THEN result2
  WHEN conditionN THEN resultN
  ELSE
END (AS ` ')
From
The LIKE operator allows you to perform pattern matching on strings☐ Can use two wildcard characters for constructing patterns:☐ percentage (%) matches any string of 0 or more characters ☐ underscore (_) matches any single character
Select name from stu where name like 'B%'

NULL Values
☐ A field with a NULL value is a field that has no value
☐ The result of any arithmetic expression involving NULL is NULL
☐ Comparing NULL values with operators such as =, < returns an unknown result
☐ You have to use the IS NULL and IS NOT NULL operators instead

☐ You can combine the result sets of two or more queries using set operators
☐ Every select must have the same number of columns
☐ The corresponding columns must have similar data types
SELECT column_name(s) FROM table1
UNION
SELECT column_name(s) FROM table2;

All aggregate functions except COUNT(*) ignore NULL values in their input
☐ COUNT(*) returns the number of rows in the table, regardless of NULL values
☐ COUNT(column) returns the number of non-null values in the specified column



SQL EXECUTION ORDER: From, on, join, where, group by, with, having, select, distinct, order by, top

CTE is a named temporary result set that only exists for the duration of the query

```
WITH courses_per_student AS (
    -- Count the number of courses each student took
    SELECT id, COUNT(DISTINCT course_id) AS num_courses
    FROM takes
    GROUP BY id
)
-- Get the distribution of courses taken by students
SELECT num_courses, COUNT(id) AS num_students
FROM courses_per_student
GROUP BY num_courses
ORDER BY num_courses;
```

A view is a virtual table based on the results set of an SQL statement
☐ Reasons to use views:
☐ Data security – allows you to hide certain data from view of certain users
☐ e.g., some users only need to know the instructor IDs and names, but not their salaries
☐ Hiding complexity of the underlying tables
☐ A view is created with the CREATE VIEW statement:
CREATE VIEW view_name AS
SELECT column1, column2, ...
FROM table_name
WHERE condition;
A view always shows up-to-date data
☐ The database engine recreates the view whenever it is used in a query

☐ In order to execute SQL queries in Python, you need a cursor object A cursor object allows you to traverse over database records .To process the rows in the result one at a time, you can use the cursor as an iterator: The cursor has a few useful properties that provide information about the result set
☐ column_names – returns the list containing the column names of the result set ☐ rowcount – the number of rows in the result set

A parameterized query is a query that uses placeholders (%) for attribute values(solution to injection attack)
A transaction is a sequence of SQL statements that represents a single unit of workThe transaction must end with one of the following statements:
☐ COMMIT: The updates performed by the transaction become permanent in the database
☐ ROLLBACK: All updates performed by the transaction are undone transactions need to be written inside a stored procedure
DECLARE EXIT HANDLER FOR SQLEXCEPTION ROLLBACK;
START TRANSACTIONS; COMMIT;
An index is a data structure that can speed up queries/searches Causes updates to the table to become slower (since the index also needs to be updated), Storage Space
CREATE [UNIQUE] INDEX index_name ON table_name (column1, column2, ...); A clustered index defines the physical order in which table records are stored There can be only one clustered index per table. By default a clustered index is created on a primary key column
Nympy: To create a 2D array, you can pass a list of lists. A=np.array([1,2],[3,4]) **A = np.zeros((3,3)), B = np.ones(5),C = np.full ((3,3),7).** **np.random.random**(shape=(3,3), creates an array with numbers sampled randomly from the uniform distribution over [0, 1). **np.random.randint(**low, [high], size),(1,10,(5,5)) creates an array with integer numbers sampled from the interval [low, high) **np.random.normal**(loc=0.0, scale=1.0, size=(3,3)) samples numbers from the normal distribution with mean = loc and standard deviation = scale. **np.arange**([start], stop, [step]) creates an array from a sequence of numbers Its arguments are the same as in Python's range() function.
np.arange(0,2,0.2)->array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8])
np.linspace(start, stop, num) creates an array of evenly spaced numbers,np.linspace(0,5,4)->array([0. , 1.66666667, 3.33333333, 5.]).**reshape(3,4)**-> will change the di,ention to 3raws and 4 column. **ravel()** can be used to flatten a multidimensional array into one dimension: array.ravel()
Indexing: A[1,2] # raw1, column 2. A[-1,-1] # last raw and column. A[1] # all raw 1. **Slicing** A[0:2] # raw 0 to 2 not included (first two). A[:,0:2] # column 0 to 2 not include (first two). A[0:2,0:2] # the first two rows and column. a[1,1:]# the second raw and second column onwards . **Fancy indexing:** a[[0,2]] first and third raw. A[:,[1,3]] second and fourth columns. **Boolean indexing/masking** is used to select elements of the array that satisfy some condition: a[a%2==0] will return the numbers in a. a%2==0 will return true/false masking. **Changing elements**: Changes to a sliced or indexed array are reflected in the original array. a[[2,4]]=99 or a[a<5]=99-> if a number is equal to 2 or 4 it will change to 99. Comparisons are performed element-wise and produce an array of Boolean values. Logical operators: & | . **.Calculations** by row and column a.mean(axis=0). Missing values: Do not test nans for equality (np.nan == np.nan is False), instead use np.isnan(a)->result will be True/False. NaN is a bit like a virus – the result of any arithmetic with nan will also be a nan: if you have nan the result for the calculation would be non. Eg, np.sum(), np.nansum() is a safe NaN version. **np.any**(a>5,axis=0) or **np.all( )**. **np.sort(a)** - will not sort the original. a.sort()- will. Can add axis. **np.argsort()** returns the indexes that would sort an array rather than the sorted elements themselves
Searching: np.where(condition) returns the indices of all the elements that satisfy the condition.In case of a 2D array, it returns a tuple of the row and the column indices of the elements that satisfy the condition. **Unique values:** np.unique( ) -> value, count = np.unique(a, return_counts=True) + value[count>2] -> result the numbers whitch appear more then 2. **closest value in the array a**: def closee(a,v): / dis = np.abs(a-v)/ return a[np.argmin(dis)].
np.concatenate([[100],array1,[100]]) **np.append(arr, values, axis=None)** appends values to the end of an array. values must be of the correct shape (the same shape as arr, excluding axis). axis specifies the axis along which to insert values. np.append(a,[[1,2,3]], axis = 0). **np.vstack**((a,b)) stacks the arrays vertically (row-wise). **np.hstack**((a,b)) stacks the arrays horizontally (column-wise). **Broadcasting** allows operations to be performed on arrays of different shapes. Rule 1: If the two arrays have different numbers of dimensions, the shape of the smaller array is padded with ones on its left side. Rule 2: The arrays are compatible if all their dimensions are equal or one of them is 1. Rule 3: If the arrays are compatible and their shapes don't match in one of the dimensions, the array with dimension of 1 is stretched to match the other array.

np.eye(N) creates an identity matrix of size N xN. np.diag(v) creates a diagonal matrix from the 1-D array v: The transpose of a matrix results from "flipping" the rows and columns. **Did not include:** Linear Algebra, Linear Equations, Determinant, Eigenvalues and Eigenvectors) **Pandas**:Can also construct a series from a dictionary.- just insert the name of the dictionary. **df.index** gives the row labels.**df.columns** give the column labels**df.dtypes** gives the data types of the columns **df.shape** gives the axis dimensions as in NumPy. values gives the data in the DataFrame as a 2D NumPy array.We can access raws by its index or name: ser[2] or ser['b']. **Slicing**: same as accessing-> ser[0:2], ser['a':'b''] only the first two. **df.loc[]** selects a group of rows and columns using the index and column names. A single label, e.g., 'a' A list of labels, e.g., ['a', 'b']A slice object with labels, e.g., 'a':'f' A Boolean mask, e.g., 'a' > 5. **df.iloc[]** selects a group of rows and columns based on their integer positions. An integer, e.g., 5 A list of integers, e.g., [4, 3, 0] A slice object with ints, e.g., 1:7 You can use .loc and .iloc to assign values to specific cells: df.loc['age'] += 1. To **select** a single column you can also use df[column_name] or df.column_name. **d = df.loc[(df['animal']=='cat') & (df['size'] == 'L')] ... (df.loc[df.index[-1],'weight'] = 15** or **df.iloc[-1, df.columns.get_loc('weight')] = 15**) You can add a column to the DataFrame by assigning a list or a NumPy array to a new column name: df.new_column_name = [.....] **sort_index**(axis=0, ascending=True) sorts the DataFrame by its row or column index.Use axis=0 to sort by the row index and axis=1 to sort by the column names. **sort_values(**by, axis=0, ascending=True) sorts by the values along the specified axis.by is a label or list of labels to sort by. **nlargest(n, columns)** returns the n rows with the largest values in the specified columns. **isna(), isnull()** = Generate a boolean mask indicating missing values. **dropna**(axis=0, how='any') = Return a filtered version of the data. **fillna(5)** = Return a copy of the data with missing values filled or imputed.