Notebook 1

In [54]:

```
#!pip install pymysql
#!pip install --upgrade pip
#!pip install psycopg2 (for PostgreSQL databases)
#!pip install kagglehub
#!pip install psycopg2-binary ipython-sql
import psycopg2 # postgreSQL driver for python. To connect.
from sqlalchemy import create_engine, text # testx wraps a raw SQL
import os
import pandas as pd
os.environ["KAGGLEHUB_CACHE"] = "/Users/catarina/Desktop/EMDYN"
import kagglehub
import subprocess # very useful module that allows me to more elega
from IPython.display import HTML, display # like the sql, to read h
import pymongo
from bson import ObjectId  # for handling ObjectId fields if needed
from datetime import datetime
```

1. Reverts a normal list

In [55]:

```
def reverse_list(lst):
    """
    Return a new list that is the reverse of lst.
    """
    return lst[::-1]
```

In [56]:

```
# Example:
original = [1, 2, 3, 4, 5]
reversed_list = reverse_list(original)

print("Original:", original)      # Original: [1, 2, 3, 4, 5]
print("Reversed:", reversed_list) # Reversed: [5, 4, 3, 2, 1]

original = ['water', 'fire', 'land', 'air']
reversed_list = reverse_list(original)

print("Original:", original)      # Original: [1, 2, 3, 4, 5]
print("Reversed:", reversed_list) # Reversed: [5, 4, 3, 2, 1]
```

```
Original: [1, 2, 3, 4, 5]
Reversed: [5, 4, 3, 2, 1]
Original: ['water', 'fire', 'land', 'air']
Reversed: ['air', 'land', 'fire', 'water']
```

2. Write a function in Python that reverses a linked list.

A linked list is a fundamental data structure in computer science used to store a sequence of elements—called nodes—where each node holds:

A piece of data (often called its “value” or “payload”). A reference (pointer) to the next node in the sequence (and in some variants, also to the previous node).

[Node A] → [Node B] → [Node C] → None data data data next next next

Advantages and disadvantages on each list kind

Different. Use array when we need to acess entries, when we need to look up by index When we know the size and it will not increase use array. Linked list: if entries will be deleted. No need to acess random indexes.

3. Merge two sorted arrays into a single sorted array

Merge two arrays, A and B. Get the lenght of each array. Staring from i=0 j=0 Compare the values of the array for index i and j. Depending when which is larger add to the array C (A+B).

4. Describe how hash tables work, including collision resolution strategies.

Dinamic set of data They allow for insert/delete/search. Search is they best part because they take in average O(1) (a big O of 1), and a O(n) in the worst case.

NOTE

- O(1) is **constant time**, which means it doesnt take longer as the input size increases. For example, referencing an item in an array takes O(1) time.
- O(logn) is **logarithmic time**, which means as the input size increases it takes a logarithmically small amount more time. For example, binary searching a sorted list is O(logn).
- O(n) is **linear time**, which means it takes a constant factor of time proportional to the size of the input size. For example, iterating over every element of an array 5 times is O(n).

Many times confused with a dictionary. A hash table is actually a dictionary using a hash function. Let's look at direct-acess tables, similar concept to hash tables. They are actually an array. Constant time operations. If we need to store an infinite number of heys, the iniverse is **unbounded and imprатical** to store in memory.

In the hash tables the universe is not unbounded, the space is O(k). There is a hash function that maps keys to a location int the table that has data. There may be two keys with the same data. This **collision** can be dealt with ccaining. this is creating list within the data. The best way is to prevent collisions. A way to do it is with the division. By doing this you create a table with size m and spread you data through the table, with a division. The index of each data stores in the table is obtained by looking at the remainder between the value of the data and the size of the table.

In [57]:

```
# Dictonary example
dictionary = {
    'a': 1,
    'b': 9,
    'c': 'C',
    'd': True
}

# insert
dictionary['e'] = False

# delete
del dictionary['a']

# search
print(dictionary['c'])
```

```
C
```

5. Write a SQL query to find the second highest salary from a table of employee salaries.

I want to use PostgreSQL.

It is a database management system (RDBMS). OIt is a software. It allows to store, organize using SQL language. Other alternative could be MySQL and IBM db2.

In [58]:

```
# download dataset
path = kagglehub.dataset_download("hummaamqaasim/jobs-in-data")
print("Path to dataset files:", path)
print(os.listdir(path)[0])

file=path+'/'+(os.listdir(path)[0])
print(file)
```

```
Path to dataset files: /Users/catarina/Desktop/EMDYN/datasets/hummaa
mqaasim/jobs-in-data/versions/6
jobs_in_data.csv
/Users/catarina/Desktop/EMDYN/datasets/hummaamqaasim/jobs-in-data/ve
rsions/6/jobs_in_data.csv
```

In [59]:

```
# read the file into pandas
df = pd.read_csv(file)

# show first rows
print("Loaded CSV with shape:", df.shape)
df.head()
```

```
Loaded CSV with shape: (9355, 12)
```

Out[59]:

	work_year	job_title	job_category	salary_currency	salary	salary_in_us
0	2023	Data DevOps Engineer	Data Engineering	EUR	88000	950
1	2023	Data Architect	Data Architecture and Modeling	USD	186000	1860
2	2023	Data Architect	Data Architecture and Modeling	USD	81800	818
3	2023	Data Scientist	Data Science and Research	USD	212000	2120
4	2023	Data Scientist	Data Science and Research	USD	93300	933

In [60]:

```
df.describe(include='all')
```

Out[60]:

	work_year	job_title	job_category	salary_currency	salar
count	9355.000000	9355	9355	9355	9355.0000
unique	NaN	125	10	11	Na
top	NaN	Data Engineer	Data Science and Research	USD	Na
freq	NaN	2195	3014	8591	Na
mean	2022.760449	NaN	NaN	NaN	149927.9812
std	0.519470	NaN	NaN	NaN	63608.8353
min	2020.000000	NaN	NaN	NaN	14000.0000
25%	2023.000000	NaN	NaN	NaN	105200.0000
50%	2023.000000	NaN	NaN	NaN	143860.0000
75%	2023.000000	NaN	NaN	NaN	187000.0000
max	2023.000000	NaN	NaN	NaN	450000.0000

In [61]:

```
!brew services start postgresql
!pg_isready
%load_ext sql
```

```
Service `postgresql@14` already started, use `brew services restart
postgresql@14` to restart.
/tmp:5432 - accepting connections
```

In [62]:

```python
# Function to help with the bash commands, using the subprocess mod
def run_shell_command(cmd):
    result = subprocess.run(cmd, capture_output=True, text=True)
    return result

# Create mydata database, or say if it already exists
db_check = run_shell_command([
    "psql", "-U", "postgres", "-tAc",
    "SELECT 1 FROM pg_database WHERE datname='mydata';"
])
if db_check.stdout.strip() == "1":
    print("Database 'mydata' already exists.")
else:
    create_db = run_shell_command(["createdb", "employees"])
    if create_db.returncode == 0:
        print("Database 'mydata' created successfully.")
    else:
        print("Error creating database 'mydata':", create_db.stderr

# Create my username or say if already exists
user_check = run_shell_command([
    "psql", "-U", "postgres", "-tAc",
    "SELECT 1 FROM pg_roles WHERE rolname='cbranco';"
])
if user_check.stdout.strip() == "1":
    print("User 'cbranco' already exists.")
else:
    create_user = run_shell_command([
        "psql", "-U", "postgres", "-c",
        "CREATE USER cbranco WITH PASSWORD '0000';"
    ])
    if create_user.returncode == 0:
        print("User 'cbranco' created successfully.")
    else:
        print("Error creating user 'cbranco':", create_user.stderr)

# 3) Grant privileges on 'mydata' to 'cbranco'
grant_privs = run_shell_command([
    "psql", "-U", "postgres", "-c",
    "GRANT ALL PRIVILEGES ON DATABASE mydata TO cbranco;"
])
if grant_privs.returncode == 0:
    print("Granted all privileges on 'mydata' to 'cbranco'.")
else:
    print("Error granting privileges:", grant_privs.stderr)
```

```
Database 'mydata' already exists.
User 'cbranco' already exists.
Granted all privileges on 'mydata' to 'cbranco'.
```

In [63]:

```python
# Reuse your Postgres credentials
DB_USER     = "cbranco"
DB_PASSWORD = "0000"
```

```
DB_HOST     = "localhost"
DB_PORT     = "5432"
DB_NAME     = "mydata"

engine = create_engine(
    f"postgresql+psycopg2://{DB_USER}:{DB_PASSWORD}@{DB_HOST}:{DB_P
)

#send the df to sql table (employees)
df.to_sql(
    name="employees",
    con=engine,
    if_exists="replace",
    index=False
)

print("✅ CSV has been written into Postgres as table `employees`."
```

```
✅ CSV has been written into Postgres as table `employees`.
```

In [64]:

```
# Read back the first 5 rows from Postgres
df2 = pd.read_sql_query("SELECT * FROM employees LIMIT 5;", con=eng
df2
```

Out[64]:

	work_year	job_title	job_category	salary_currency	salary	salary_in_us
0	2023	Data DevOps Engineer	Data Engineering	EUR	88000	950
1	2023	Data Architect	Data Architecture and Modeling	USD	186000	1860
2	2023	Data Architect	Data Architecture and Modeling	USD	81800	818
3	2023	Data Scientist	Data Science and Research	USD	212000	2120
4	2023	Data Scientist	Data Science and Research	USD	93300	933

In [65]:

```
sql_snd_dalary = """

SELECT
  salary_in_usd
FROM (
  SELECT
    salary_in_usd,
    DENSE_RANK() OVER (ORDER BY salary_in_usd DESC) AS rnk
  FROM
    employees
) AS ranked_salaries
WHERE
  rnk = 2;
```

```
"""
```

```
In [66]: snd_salary = pd.read_sql_query(sql_snd_dalary, con=engine)
```

```
In [67]: print(snd_salary)
```

```
   salary_in_usd
0         430967
```

```
In [68]: # check which country pays the most
         # to run the sql query, save it to a variable

         sql_highest_salary="""
         SELECT
           company_location,
           ROUND(AVG(salary_in_usd)::numeric, 0) AS avg_salary,
           COUNT(*) AS num_jobs
         FROM
           employees
         GROUP BY
           company_location
         ORDER BY
           avg_salary DESC
         LIMIT 10;
         """
```

```
In [69]: highest_salary = pd.read_sql_query(sql_highest_salary, con=engine)
```

```
In [70]: print(highest_salary)
```

```
         company_location  avg_salary  num_jobs
0                   Qatar    300000.0         1
1             Puerto Rico    167500.0         4
2                   Japan    165500.0         4
3           United States    158159.0      8132
4                  Canada    143919.0       226
5            Saudi Arabia    134999.0         2
6               Australia    132283.0        24
7             New Zealand    125000.0         1
8                 Ukraine    121333.0         6
9  Bosnia and Herzegovina    120000.0         1
```

6. Differences between INNER JOIN, LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN.

- **INNER JOIN** returns only the rows for which there is a match in both tables. In other words, it “intersects” Table A and Table B on the join key(s).
- **LEFT JOIN**, returns all rows from the *left* table (A), plus matched rows from the *right* table (B). If there is no match in B, you still get all the A-rows, but any columns from B will be NULL.

- **RIGHT JOIN**, returns all rows from the *right* table (B), plus matched rows from the *left* table (A). If there is no match in A, you still get all the B-rows, but any columns from A become NULL.
- **FULL OUTER JOIN**, returns all rows from both tables (A and B). If a row in A has a match in B, you combine them. If a row in A has no match in B, you still see the A-row with NULLs for B's columns. If a row in B has no match in A, you see the B-row with NULLs for A's columns. In practice, you see A∩B (the overlap), plus A-only, plus B-only.

7. Design a database schema for a social media application's "follow" relationships?

In [71]:

```
# create a table for the social media application users and followe
# with the """ I can write exactly how I would write in an SQL quer
create_users_followers_tables_sql = """
-- Create users table
CREATE TABLE IF NOT EXISTS users (
  id              BIGSERIAL     PRIMARY KEY,
  username        VARCHAR(50)   NOT NULL UNIQUE,
  email           VARCHAR(255)  NOT NULL UNIQUE,
  created_at      TIMESTAMP     NOT NULL DEFAULT NOW(),
  password_hash   VARCHAR(255)  NOT NULL,
  CHECK (username <> '')
);

-- Create follows table
CREATE TABLE IF NOT EXISTS follows (
  follower_id BIGINT    NOT NULL,
  followee_id BIGINT    NOT NULL,
  created_at  TIMESTAMP NOT NULL DEFAULT NOW(),

  PRIMARY KEY (follower_id, followee_id), -- single combinations, p

  FOREIGN KEY (follower_id)
    REFERENCES users (id)
    ON DELETE CASCADE,
  FOREIGN KEY (followee_id) -- if a user is deleted the whole casca
    REFERENCES users (id)
    ON DELETE CASCADE,

  CHECK (follower_id <> followee_id)
);

-- Indexes to speed up lookups
CREATE INDEX IF NOT EXISTS idx_follows_follower ON follows (followe
CREATE INDEX IF NOT EXISTS idx_follows_followee ON follows (followe
"""

# Execute the DDL in PostgreSQL
with engine.begin() as conn:
    conn.execute(text(create_users_followers_tables_sql))
```

```
print("✅ Tables `users` and `follows` have been created (or alread
```

```
✅ Tables `users` and `follows` have been created (or already existe
d).
```

In [72]:

```
# create some mockup data
insert_users_sql = """
INSERT INTO users (username, email, password_hash)
VALUES
  ('alice',   'alice@example.com',   '$2b$12$abcdefghijk01234567890
  ('bob',     'bob@example.com',     '$2b$12$mnopqrstuv345678901234
  ('carol',   'carol@example.com',   '$argon2i$v=19$m=65536,t=2,p=1
  ('dave',    'dave@example.com',    '$argon2i$v=19$m=65536,t=2,p=1
  ('erin',    'erin@example.com',    '$2b$12$uvwxyz9876543210123456
ON CONFLICT (username) DO NOTHING;
"""

with engine.begin() as conn:
    conn.execute(text(insert_users_sql))

print("✅ Sample rows inserted into `users`.")
```

```
✅ Sample rows inserted into `users`.
```

In [73]:

```
# get the users id, so that then I can fake follow events.
with engine.connect() as conn:
    users_df = pd.read_sql("SELECT id, username FROM users ORDER BY

print("\nCurrent users (id ↔ username):")
print(users_df.to_string(index=False))

# build a dictionary where the username is teh dictionary keym and
user_id_map = dict(zip(users_df["username"], users_df["id"]))

# follow-rows based on those IDs
follows_rows = [
    (user_id_map["alice"], user_id_map["bob"]),
    (user_id_map["alice"], user_id_map["carol"]),
    (user_id_map["bob"],   user_id_map["alice"]),
    (user_id_map["carol"], user_id_map["dave"]),
    (user_id_map["carol"], user_id_map["erin"]),
    (user_id_map["erin"],  user_id_map["alice"]),
]

# insert them with “ON CONFLICT DO NOTHING” to avoid duplicates
insert_follows = text("""
INSERT INTO follows (follower_id, followee_id)
VALUES (:follower_id, :followee_id)
ON CONFLICT (follower_id, followee_id) DO NOTHING;
""")

with engine.begin() as conn:
    for follower_id, followee_id in follows_rows:
        conn.execute(insert_follows, {"follower_id": follower_id, "

print("\n✅ Sample rows inserted into `follows`.")
```

```
Current users (id ↔ username):
 id username
  1    alice
  2      bob
  3    carol
  4     dave
  5     erin

✅ Sample rows inserted into `follows`.
```

In [74]:

```python
# show
with engine.connect() as conn:
    users_full = pd.read_sql("SELECT * FROM users ORDER BY id;", co
    follows_full = pd.read_sql("""
        SELECT
            f.follower_id,
            u1.username AS follower_username,
            f.followee_id,
            u2.username AS followee_username,
            f.created_at
        FROM follows AS f
        JOIN users u1 ON u1.id = f.follower_id
        JOIN users u2 ON u2.id = f.followee_id
        ORDER BY f.follower_id, f.followee_id;
    """, conn)

print("\n--- `users` table: ---")
print(users_full.to_string(index=False))

print("\n--- `follows` table (joined with usernames): ---")
print(follows_full.to_string(index=False))
```

```
--- `users` table: ---
 id username             email                 created_at
password_hash
  1    alice alice@example.com 2025-06-04 12:11:55.737032   $2b$12$a
bcdefghijk0123456789012345678901234567890
  2      bob   bob@example.com 2025-06-04 12:11:55.737032 $2b$12$mno
pqrstuv3456789012345678901234567890123456
  3    carol carol@example.com 2025-06-04 12:11:55.737032    $argon2
i$v=19$m=65536,t=2,p=1$abcd1234$xyz9876543210abcdef
  4     dave  dave@example.com 2025-06-04 12:11:55.737032  $argon2i$
v=19$m=65536,t=2,p=1$wxyz5678$lmn543210abcdef987654
  5     erin  erin@example.com 2025-06-04 12:11:55.737032  $2b$12$uv
wxyz98765432101234567890123456789012345678901234567

--- `follows` table (joined with usernames): ---
 follower_id follower_username  followee_id followee_username
created_at
           1             alice            2               bob 2025-0
6-04 12:20:00.757452
           1             alice            3             carol 2025-0
6-04 12:20:00.757452
           2               bob            1             alice 2025-0
6-04 12:20:00.757452
           3             carol            4              dave 2025-0
6-04 12:20:00.757452
           3             carol            5              erin 2025-0
6-04 12:20:00.757452
           5              erin            1             alice 2025-0
6-04 12:20:00.757452
```

In [75]:

```
# who is'alice' following
alice_id = user_id_map['alice']
following_df = pd.read_sql(text("""
    SELECT
        u.id AS user_id,
        u.username AS username,
        f.created_at AS followed_at
    FROM follows f
    JOIN users u ON u.id = f.followee_id
    WHERE f.follower_id = :alice_id
    ORDER BY u.username;
"""), engine, params={"alice_id": alice_id})

print("Users Alice is following:")
print(following_df.to_string(index=False))
```

```
Users Alice is following:
 user_id username                followed_at
       2      bob 2025-06-04 12:20:00.757452
       3    carol 2025-06-04 12:20:00.757452
```

8. Describe normalization and denormalization. Give an example where denormalization is preferable.

1. **Normalization** is the process of organizing a relational database so that:

- Redundancy is minimized, and
- Data integrity is ensured through well-defined relationships. The core idea is to split data into multiple related tables and enforce relationships via foreign keys, thereby avoiding duplicate storage of the same information.

2. **Denormalization** is the intentional introduction of redundancy—combining or pre-joining tables, or storing computed/duplicated fields—so that read queries require fewer (or simpler) joins. It is usefull if we are dealing with high-traffic web pages that have a large number of reads very quickly. Yet, it can lead to redudant data (same information in multiple places).

- Preferably when a lot is requested: for instance in a social feed that shows user + profile picture + follower count. Under a normalized schema we would need must more requestes because it would be three different columns.

9. NoSQL databases

Popular NoSQL products: MongoDB (document suited), Redis (key-values) and Cassandra. No-relation database are more relaxed and flexible. They don't store the data in rigid tables, they can save in documents (similar to rowns in SQL), key-values and other. More suited for rapidly changing data, and this systems ofetn encourage denormalization, with nested or duplicated fields.
NoSQL -> Better for many many read and write requests. Horizontal and vertical scaling. SQL -> Vertical scalling. Need of relations. If the data changes a lot, and needs to be updated often.

In [76]:

```
# To delete documents of the collection
result = collection.delete_many({})
```

In [77]:

```
# connect
MONGO_URI = "mongodb://localhost:27017"
client = pymongo.MongoClient(MONGO_URI)

# create a database and a collection
db = client["my_database"]                # creates my_database
collection = db["users_collection"]   # creates users_collection

# insert a few sample documents
sample_docs = [
    {
        "username": "alice",
        "email": "alice@example.com",
        "profile": {
            "first_name": "Alice", # no age
            "last_name": "Johnson"
            # no age in this profile
        },
        "adress": "USA", # only document with this field
        "roles": ["admin", "editor"],
        "created_at": pd.Timestamp("2025-06-05 09:00:00")
    },
    {
        "username": "bob",
        "email": "bob@example.com",
        "profile": {
            "first_name": "Bob",
            "last_name": "Smith",
            "age": 28
        },
        "roles": ["editor"],
        "created_at": pd.Timestamp("2025-06-04 14:30:00")
    },
    {
        "username": "carol",
        "email": "carol@example.com",
        "profile": {
            "first_name": "Carol",
            "last_name": "Williams",
            "age": 25
        },
        "roles": ["viewer"],
        "created_at": pd.Timestamp("2025-06-02 16:15:00")
    }
]

# insert if collection is empty (so re-running doesn’t duplicate)
if collection.count_documents({}) == 0:
    collection.insert_many(sample_docs)
    print("Inserted sample documents into 'users_collection'.")
```

```
Inserted sample documents into 'users_collection'.
```

In [78]:

```
# find() on a collection searches from a document where the field r
results_cursor = collection.find({"roles": "editor"})
results_list = list(results_cursor) # a list whose elemnsta are dic
# print(results_list)
for doc in results_list:
    # Convert ObjectId to string for nicer display
    doc["_id"] = str(doc["_id"])
# print(results_list) # Improved formatting
users_df = pd.DataFrame(results_list)

print("Documents where roles include 'editor':")
display(users_df)
print(users_df)
```

```
Documents where roles include 'editor':
```

	_id	username	email	profile	ad
0	6841aa6366a7f9e0aa33040a	alice	alice@example.com	{'first_name': 'Alice', 'last_name': 'Johnson'}	
1	6841aa6366a7f9e0aa33040b	bob	bob@example.com	{'first_name': 'Bob', 'last_name': 'Smith', 'a...	

```
                         _id username              email  \
0  6841aa6366a7f9e0aa33040a    alice  alice@example.com   
1  6841aa6366a7f9e0aa33040b      bob    bob@example.com   

                                             profile adress  
roles  \
0    {'first_name': 'Alice', 'last_name': 'Johnson'}    USA  [admin, 
editor]   
1  {'first_name': 'Bob', 'last_name': 'Smith', 'a...    NaN  
[editor]   

           created_at  
0 2025-06-05 09:00:00  
1 2025-06-04 14:30:00  
```

In [79]:
```
# Who is older than 26?
cursor = collection.find({"profile.age": {"$gt": 26}}).sort("profil
df_over_26 = pd.DataFrame(list(cursor))
df_over_26["_id"] = df_over_26["_id"].astype(str)  # stringify the 
display(df_over_26)
```

	_id	username	email	profile	r
0	6841aa6366a7f9e0aa33040b	bob	bob@example.com	{'first_name': 'Bob', 'last_name': 'Smith', 'a...	[ed

In [80]:
```
# Add a new field to only one document
collection.update_one(
    {"username": "bob"},
    {"$set": {"last_login": datetime.utcnow()}}
)
```

Out[80]:
```
UpdateResult({'n': 1, 'nModified': 1, 'ok': 1.0, 'updatedExistin
g': True}, acknowledged=True)
```

In [81]:
```
# Delete one document based on username
collection.delete_one({"username": "carol"})
print("Deleted Carol’s document, if it existed.")
```

```
Deleted Carol’s document, if it existed.
```

In [82]:
```
# Show all documents
```

```python
docs = list(collection.find({}))
for d in docs:
    d["_id"] = str(d["_id"])
df = pd.DataFrame(docs)
display(df)
```

	_id	username	email	profile	ad
0	6841aa6366a7f9e0aa33040a	alice	alice@example.com	{'first_name': 'Alice', 'last_name': 'Johnson'}	
1	6841aa6366a7f9e0aa33040b	bob	bob@example.com	{'first_name': 'Bob', 'last_name': 'Smith', 'a...	

```
In [ ]:
```