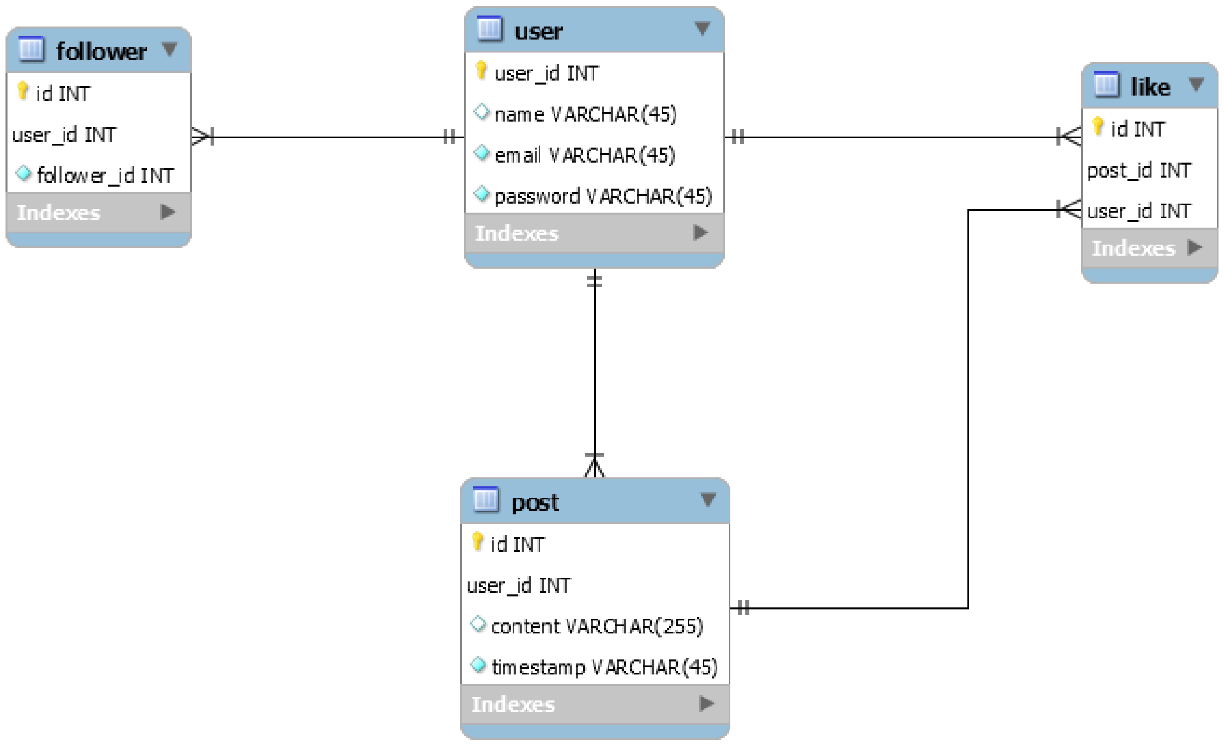
1. Design database schema for Instagram

   Solution:

   (Database Model file is attached to the mail)



2. Let's suppose we need to search across 100s of documents (search for words). One way to do it is to go one file at a time. But that will take too long. So write a multi-threaded or multi-process program to run the search across 100s of files. Keep the number of threads/processes configurable.

   Output - files which have the word being searched.

   Solution:

In python, This task can be done by assigning a number of worker threads/ processes (normally equal to the number of CPU cores). These worker threads assign the task to CPU core as per FIFO scheduling.

Python Code:

```
import time
from functools import partial
from multiprocessing import Pool, cpu_count, Process
from queue import Queue

def searchInFile(word, file):
    found = 0
    with open(file, 'r') as f:
        for line in f:
            if word in line:
                found |= 1
    return file, found

def naive(func, files):
    result = []
    for filename, found in [search(file) for file in files]:
        if found:
            result.append(filename)
    print(result)

def multiProcessing(func, files, numProcesses):
    q = Queue()
    with Pool(numProcesses or cpu_count()+1) as pool:
        for filename, found in pool.imap_unordered(func, files):
            if found:
                q.put(filename)
    print(list(q.get(1) for i in range(q.qsize())))

if __name__=="__main__":
    #startTime = time.time()
    n = 0 # number of worker process
    files = ["1.py", "2.py", "3.py", "4.py", "5.py"]
    word = "cnt"
    search = partial(searchInFile, word)

    # naive(search, files)
```

```
        multiProcessing(search, files, n)
        #print(time.time()-startTime)
```

3. Problem Statement

You are a big bank and you are making a list of the most financially reliable customers. This is going to be a sorted list and with most reliable customers in the end. Now there are multiple criteria for determining the overall reliability and all of them must be used according to their priorities. Following are the criteria:

1. Total Account Balance - The more the better
2. Age - The less the better
3. Salary - The more the better
4. Number of dependents - The less the better

These criteria have the priority as mentioned above, i.e. Total account balance will take priority over age. So for example, P1 with age 25 and 100 account balance will be placed higher than P2 with age 22 and account balance 80 but if both had the same account balance then P2 will be placed higher. The same way, all the 4 criteria will come into picture in sorting all the people accordingly.

Now you are given a list of objects with each object representing a person and has the above attributes. Your task is to sort them.

Solution:

Can be sorted using a custom comparator.

Python Code:

```
class Person:
    def __init__(self, totalAccountBalance=0, age=0, salary=0, numOfDependents=0):
        self.totalAccountBalance = totalAccountBalance
        self.age = age
        self.salary = salary
        self.numOfDependents = numOfDependents

    def keyToCompare(self):
        return [self.totalAccountBalance, self.age, self.salary, self.numOfDependents]

    def __repr__(self):
        return str(self.keyToCompare())

if __name__=="__main__":
    listOfPersons = [Person(0,1,1,2), Person(5,2,5,2), Person(2,42,4,2), Person(0,3,4,2)]
    listOfPersons.sort(key = lambda x: x.keyToCompare())
    print(listOfPersons)
```

<u>Java Code:</u>

```
// Person.java
public class Person implements Comparable<Person>{
    public int totalAccountBalance;
    public int age;
    public int salary;
    public int numOfDependents;

    public Person(int totalAccountBalance,int age, int salary, int numOfDependents){
        this.totalAccountBalance = totalAccountBalance;
        this.age = age;
        this.salary = salary;
        this.numOfDependents = numOfDependents;
    }

    @Override
    public int compareTo(Person other){
        if (this.totalAccountBalance-other.totalAccountBalance!=0)
            return this.totalAccountBalance-other.totalAccountBalance;
        if (this.age-other.age!=0)
            return this.age-other.age;
        if (this.salary-other.salary!=0)
```

```
            return this.salary-other.salary;
        return this.numOfDependents - other.numOfDependents;
    }

    @Override
    public String toString(){
        return String.format("total Account Balance: %s, age: %s, salary: %s, number Of
Dependents: %s", totalAccountBalance, age, salary, numOfDependents);
    }
}


// Main.java
import java.util.*;
public class Main{
    public static void main(String[] args){
        List<Person> listOfPersons = new ArrayList<Person>();
        listOfPersons.add(new Person(0,1,1,2));
        listOfPersons.add(new Person(5,2,5,2));
        listOfPersons.add(new Person(2,42,4,2));
        listOfPersons.add(new Person(0,3,4,2));

        Collections.sort(listOfPersons);

        for(Person person: listOfPersons){
            System.out.println(person);
        }

    }
}
```