2021

Project work I.

Half-yearly task documentation

BODROGI LILI

TABLE OF CONTENTS

Introduction 3

The theme of the project 3

How the service on which the project is based works 3

Source of data 3

Planning 4

ER-Modell 4

Tables and stored data in the database 5

People 5

Jobs 5

RockYou 5

AshleyMadison 5

LinkedIn 5

Plus 6

Date 6

Type 6

City 6

Country 6

Final tables and their relationships 7

Realization 8

Upload tables with data 8

Country 8

City 8

At_Type 8

At_Date 8

Plus 8

LinkedIn 8

AshleyMadison 9

RockYou 9

Jobs 9

People 9

Create other features 9

Sequences 9

Stored procedures 10

Triggers 11

Analyzing queries, optimizing 12

A city-by-city earner under the age of 25 who earns more than 80k? 12
Analysis 12
Optimizations 13
When did you manage to crack passwords that are rarer than average? 13
Analysis 13
Optimization 14
NoSQL Database Management 15
Convert tables in the database 15
Transfer data to MongoDB 15
Queries 17
1. How many people have a job above the average salary of €100,000? 17
2. Which city has the most employees and how many? 18
3. How many women and how many men are registered from cities of more than 10 million people? 18
4. How many women have used passwords that are less than 50% common on the dating site? 18
5. How many people are registered in the city with the smallest population? 19
6. What are the top 2 most successful types of hacking? 19
7. How many people earn over 150,000 but use a password above 65 percent frequency in general? 19
8. Which 4 are the most common jobs and how many people do it? 20
9. Which 5 passwords were most successfully hacked? 20
10. Which 3 cities have the highest number of people earning more than 200,000? 21
Attachments 22
The script that creates tables 22
Sequence Sequence 24
Procedure Procedure 24
Trigger 25
Optimization 1: 25
Optimization 2 25
2. Query 26
3. Query 26
4. Query 26
5. Query 26
6. Query 26
7. Query 26
8. Query 27
9. Query 27
10. Query 27

INTRODUCTION

THE THEME OF THE PROJECT

This project focuses on designing a robust database for a password security application. Its core theme is to establish meaningful relationships between users, passwords, and breach data to provide comprehensive insights into password strength and common vulnerabilities.

HOW THE SERVICE - WHICH THE PROJECT IS BASED ON - WORKS

The application's database is designed to assist both companies and individuals in assessing password security. Users can check the strength of their brainstormed passwords against a continuously updated database of compromised passwords, which includes data on how frequently specific passwords have been used. To provide companies with an aggregated overview of their employees' average password strength, the database also stores anonymized data linking password usage to generalized job roles. Additionally, it records details on past page breaches, including the success rate and methods of attacks, and incorporates geographical data (city and country) for research purposes.

SOURCE OF THE DATA

RockYou data: https://www.kaggle.com/wjburns/common-password-list-rockyoutxt

Ashley Madison: https://www.pxdojo.net/2015/08/what-i-learned-from-cracking-4000.html

LinkedIn: https://github.com/afiskon/password-lists/blob/master/linkedin.txt plus I added from other databases

Plus: https://www.kaggle.com/taranvee/bruteforce-database-password-dictionaries several passwords that have already been hacked from several sides.

City list: https://www.kaggle.com/max-mind/world-cities-database?select=worldcitiespop.csv (I didn't use the whole set only 2 cities per country, sometimes 1 city). I also used the https://datahub.io/core/world-cities page and Wikipedia.

Country list: https://www.nationsonline.org/oneworld/countries_of_the_world.htm

The year of the hacking: https://en.wikipedia.org/wiki/List_of_data_breaches plus my fiction.

Hacking methods: I pulled it from the notes of my IT security class.

Blue Collar: https://www.theladders.com/career-advice/30-blue-collar-jobs-with-the-highest-salaries and https://www.businessinsider.in/slideshows/miscellaneous/30-blue-collar-jobs-with-the-highest-salaries/slidelist/71301347.cms#slideid=71301386

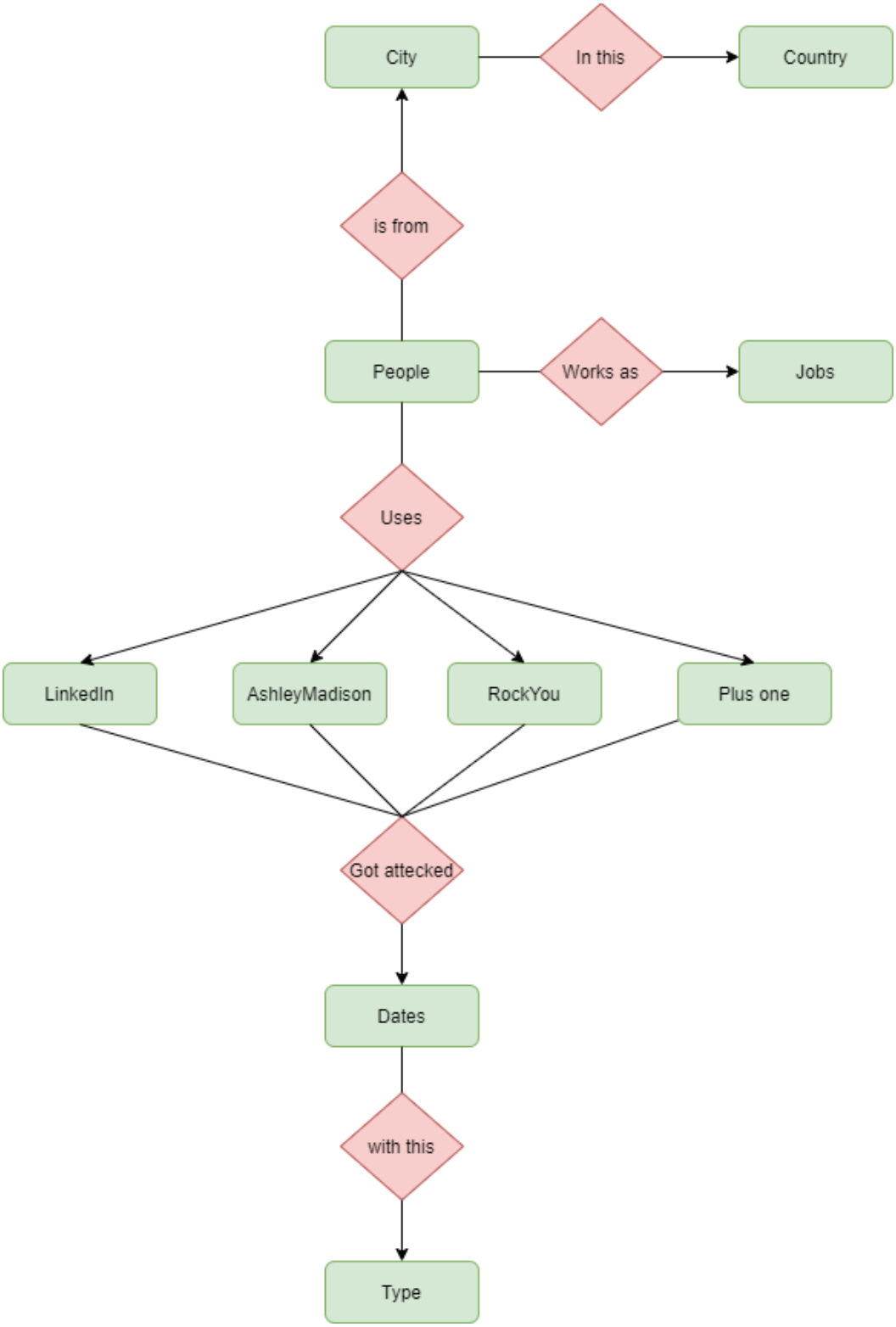
White-collar: https://www.thebalancecareers.com/best-white-collar-jobs-4159985; https://www.jobseem.com/best-white-collar-jobs-careers-meaning-examples/; Or the generator data.

I generated the rest of the data.

PLANNING

ER-MODELL

The individual relationship diagram was designed using the draw.io desktop application. This tool was selected for its intuitive interface, no-cost accessibility, and complete set of required functionalities.



TABLES AND STORED DATA IN THE DATABASE

PEOPLE

In this table, we store people who have registered on one of the 4 sites. The following data shall be stored:

- PEOPLE_ID: The number of people, will be the primary key.
- FIRST_NAME: The first name of the registered person.
- LAST_NAME: The surname of the registered person.
- BIRTH_DATE: Date of birth.
- SEX: The registered person's sex.
- EMAIL: The registered person's e-mail.
- JOB_ID: The job ID of what you worked for.
- ROCKYOU_ID: Password ID of which one they used.
- ASHLEY_ID: Password ID of which one they used.
- LINKEDIN_ID: Password ID of which one they used.
- PLUS_ID: Password ID of which one they used.
- CITY_ID: Identification of the city from which it originates.

JOBS

In this table, I store the works and the data about them. The following data shall be stored:

- JOB_ID: The job ID, it is a primary key.
- NAME: The name of the work.
- AVG_SALARY: Average salary.
- DIFFICULTY: The difficulty of work.
- KINDOF: (blue or white) Physical or mental work.

ROCKYOU

In this board, I store passwords that have ever been hacked from RockYou.

The following data is stored:

- PASSWORD_ID: Password ID, primary key.
- PASSWORD: The password that has been hacked.
- FREQUENCY: How common password.
- ATTACK_ID: Identification of the attack, foreign key.

ASHLEYMADISON

In this board, I store passwords that have ever been hacked from the dating site Ashley Madison.

The following data is stored:

- PASSWORD_ID: Password ID, primary key.
- PASSWORD: The password that has been hacked.
- FREQUENCY: How common password.
- ATTACK_ID: Identification of the attack, foreign key.

LINKEDIN

In this table, I store passwords that have ever been hacked from LinkedIn.

The following data is stored:

- PASSWORD_ID: Password ID, primary key.
- PASSWORD: The password that has been hacked.
- FREQUENCY: How common password.
- ATTACK_ID: Identification of the attack, foreign key.

PLUS

In this table, I store passwords that have ever been hacked from any side.

The following data is stored:

- PASSWORD_ID: Password ID, primary key.
- PASSWORD: The password that has been hacked.
- FREQUENCY: How common password.
- ATTACK_ID: Identification of the attack, foreign key.

DATE

In this table are stored the times of attacks from 1995 to the present day.

The following data is stored:

- ATTACK_ID: The identity of the breakup, primary key.
- TYPE_ID: What type they tried, foreign key.
- CRACKED_PASSW: How many pieces have been hacked.
- SUCCESSFUL: Whether it was successful, or only half, or failed.
- Year: When they tried.

TYPE

In this table are stored the types of attacks.

The following data is stored:

- TYPE_ID: breakup ID, primary key.
- NAME: The name of the method of hacking.
- DIFFICULTY: Difficulty level.

CITY

In this table are stored the cities from which people registered.

The following data is stored:

- NAME: The name of the city.
- CITY_ID: City ID, primary key.
- POPULATION: Population of the city.
- COUNTRY_ID: The id of the country to which it belongs is a foreign key.

COUNTRY

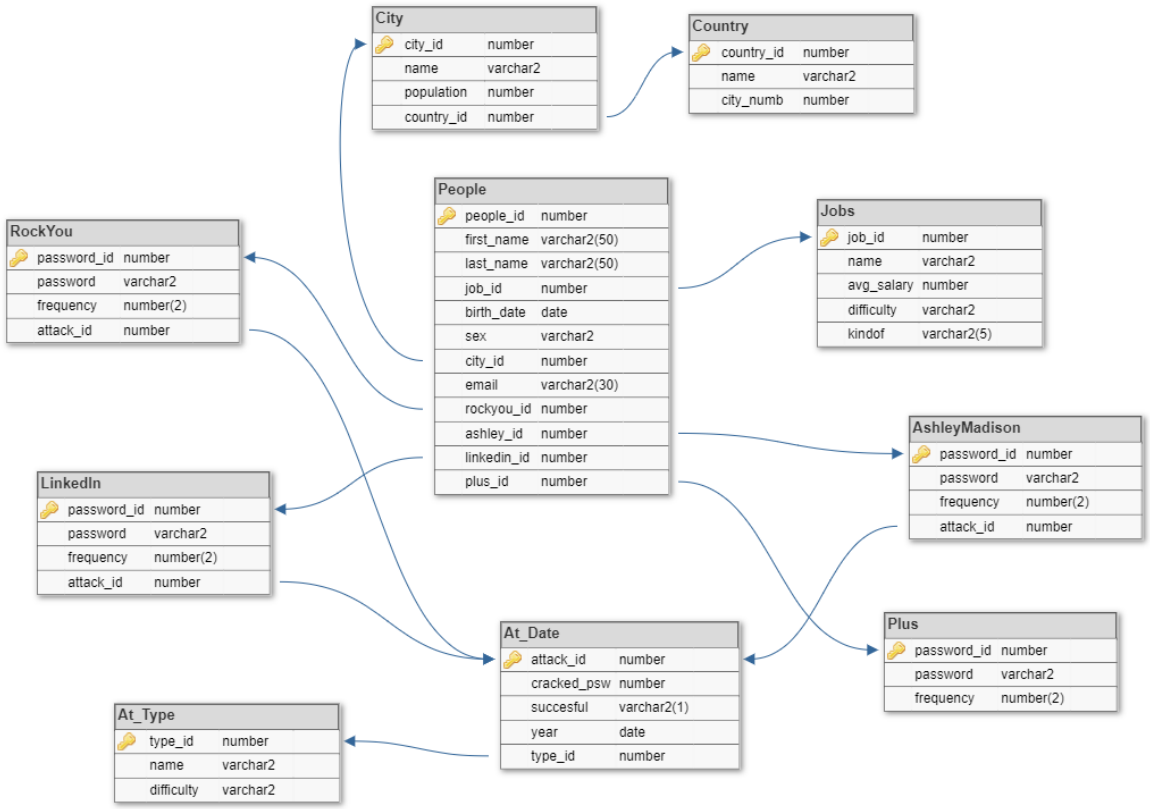
In this table are stored the countries from which people are registered.

The following data is stored:

- COUNTRY_ID: COUNTRY ID, primary key.
- NAME: The name of the country.
- CITY_NUM: How many cities are there in the country.

FINAL TABLES AND THEIR RELATIONSHIPS

After designing[1] the tables, the database became the final model:



1 The script that created the tables can be found in the annex at the end of the table of contents.

CONTENT

After designing and creating the tables, I created the necessary data with which I filled them, as well as created other necessary functions. (Sequence, trigger)

UPLOAD TABLES WITH DATA

In this chapter, I'll describe how the data was created and how many records the table contains.

COUNTRY

- Since I found a page from which I could put the data in an excel spreadsheet, so, filtering it to larger countries, I used this excel database. I uploaded the IDs using a sequencing method.
- **There were 49** data.

CITY

- Using the pages, I created an excel that generally stores the 2 most populous cities in the country. (If it's a small country, it's only 1.) Since I created IDs with sequence, I generated insert methods with Oracle from the CSV file and then added sequence.
- **95** data

AT_TYPE

- Since there is not much data in this table, I wrote them manually. The IDs, of course, were also created by sequence.
- **There** were 5 data.

AT_DATE

- At first, I easily analyzed the data with **dbForge for Oracle**, and then I updated the successful data so that where the password number is 0, the success cannot be "yes".
- **50,000** date data.

PLUS

- I used the downloaded txt file in **dbForge for Oracle** and generated ID and frequency for them.
- **523294** data.

LINKEDIN

- I used the downloaded txt file in dbForge for Oracle and generated ID and frequency for them.
- **it became 471688** data.

ASHLEYMADISON

- I used the downloaded txt file in **dbForge for Oracle** and generated ID and frequency for them.
- A total of **4,007** pieces.

ROCKYOU

- I used only a small part of the originally hacked password txt, as the original consists of more than 10 million passwords.
- Thus, the shortened list consists of **517710** passwords.

JOBS

- Jobs that people do. I took these into two groups, white-collar and blue-collar. Since I didn't need a lot of data, I took some of them off the internet and wrote a few by hand. But in the end, I used **dbForge for Oracle** to generate SQL.
- A total of **184** data were obtained.

PEOPLE

- **I used dbForge for Oracle** to generate the data. Then I modified the resulting SQL file and uploaded the IDs with a sequence, while the email addresses remained empty for the time being because I did it with a stored procedure after uploading.
- **87,600** data

CREATE OTHER FEATURES

SEQUENCES

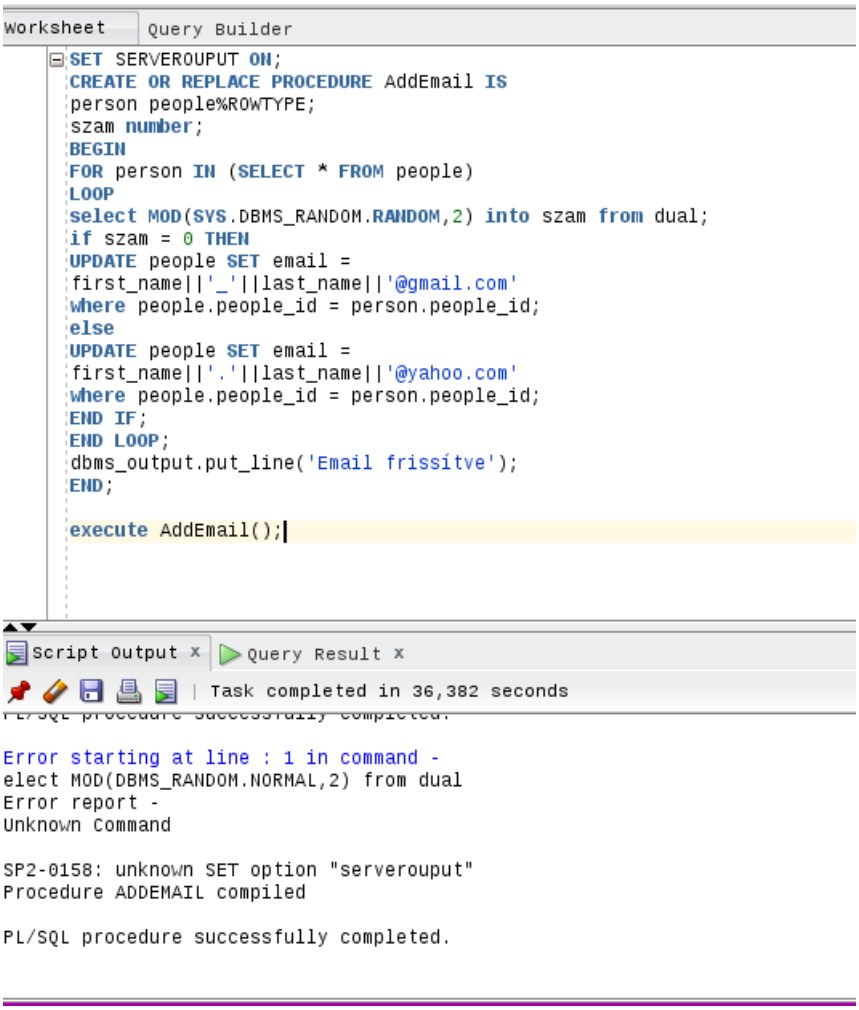
To generate primary keys more easily later, I created sequences for some tables. The tables with the sequence are therefore as follows:

- City
- At_type
- Jobs
- People

The script that creates the sequences can be found in the annex.

STORED PROCEDURES

I created this stored procedure because I left it empty when uploading because I wanted to adapt the email address to the names.



```
Worksheet  Query Builder
SET SERVEROUPUT ON;
CREATE OR REPLACE PROCEDURE AddEmail IS
person people%ROWTYPE;
szam number;
BEGIN
FOR person IN (SELECT * FROM people)
LOOP
select MOD(SYS.DBMS_RANDOM.RANDOM,2) into szam from dual;
if szam = 0 THEN
UPDATE people SET email =
first_name||'_'||last_name||'@gmail.com'
where people.people_id = person.people_id;
else
UPDATE people SET email =
first_name||'.'||last_name||'@yahoo.com'
where people.people_id = person.people_id;
END IF;
END LOOP;
dbms_output.put_line('Email frissítve');
END;

execute AddEmail();
```

```
Script Output x  Query Result x
Task completed in 36,382 seconds

Error starting at line : 1 in command -
elect MOD(DBMS_RANDOM.NORMAL,2) from dual
Error report -
Unknown Command

SP2-0158: unknown SET option "serverouput"
Procedure ADDEMAIL compiled

PL/SQL procedure successfully completed.
```

I uploaded all the e-mails.

TRIGGERS

I've implemented a trigger that automatically generates an email address for new entries in the database if the email field is null. This trigger constructs the email using a specific format: '{firstname}_{lastname} {birth_year}'. This ensures that every new person added to the database has a populated email address.

ANALYZING QUERIES, OPTIMIZING

PEOPLE UNDER THE AGE OF 25 WHO EARNS MORE THAN 80K, CITY BY CITY?

The objective of my first query is to determine the count of earners, aged 25 or below, within each respective city, who are employed in roles where the average annual salary surpasses EUR 80,000. To achieve this, the query necessitates joining three distinct tables, among which the "People" table is considerably large, comprising approximately 900,000 entries.

```
select COUNT(p.PEOPLE_ID), c.NAME
from people p inner join city c on p.city_id=c.city_id
inner join jobs j on p.job_id=j.job_id
where j.AVG_SALARY>80000
and (EXTRACT(YEAR FROM sysdate)-EXTRACT(YEAR FROM p.BIRTH_DATE))<25
group by c.NAME;
```

ANALYSIS

OPERATION	OBJECT_NAME	CARDINALITY	COST
SELECT STATEMENT		66582	3048
HASH (GROUP BY)		66582	3048
HASH JOIN		66582	3045
Access Predicates			
P.CITY_ID=C.CITY_ID			
TABLE ACCESS (FULL)	CITY	95	3
HASH JOIN		66582	3042
Access Predicates			
P.JOB_ID=J.JOB_ID			
TABLE ACCESS (FULL)	JOBS	116	3
Filter Predicates			
J.AVG_SALARY>80000			
TABLE ACCESS (FULL)	PEOPLE	104465	3039
Filter Predicates			
EXTRACT(YEAR FROM SYSDATE@!)-EXTRACT(YEAR FRO			

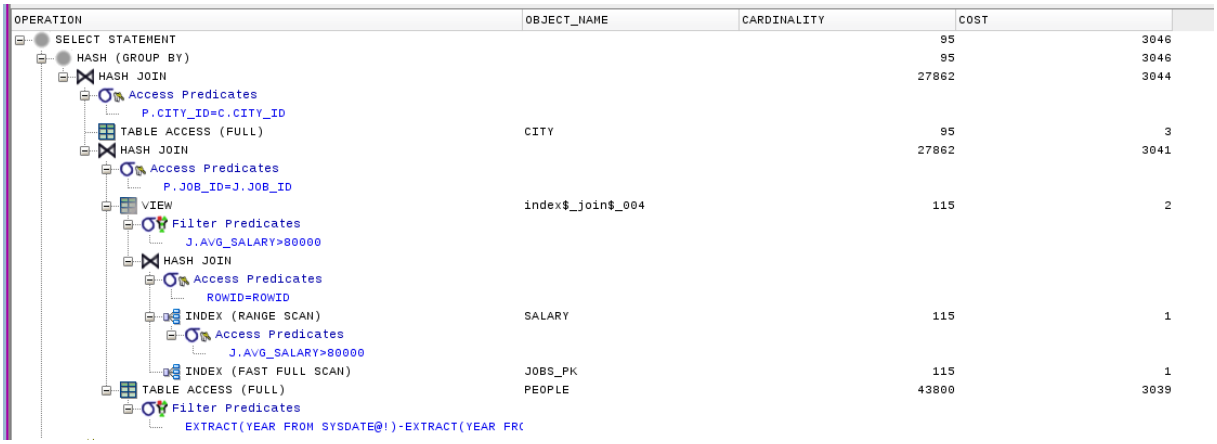
- The query execution plan initiates with a full table scan on the City table, leading into a hash join. Prior to this, a nested hash join is performed, involving full table scans on both the Jobs and City tables. Filtering conditions are applied during the full table scans of both the Jobs and People tables.
- The specific join order between City-People and Jobs-People is of minor significance due to their small data volumes. Oracle's optimizer, however, prioritizes the City table, likely because it is roughly half the size of the Jobs table. Access predicates are utilized in both hash joins to facilitate efficient table connections.
- It's notable that the estimated cardinality and cost in the plan both exceed the actual values.

OPTIMIZATIONS

To optimize the query, I created indexes:

- city_id, job_id, birth_date fields in the people table,
- and the avg_salary field in the Jobs table.

After optimization, the following implementation plan was proposed by the database manager:



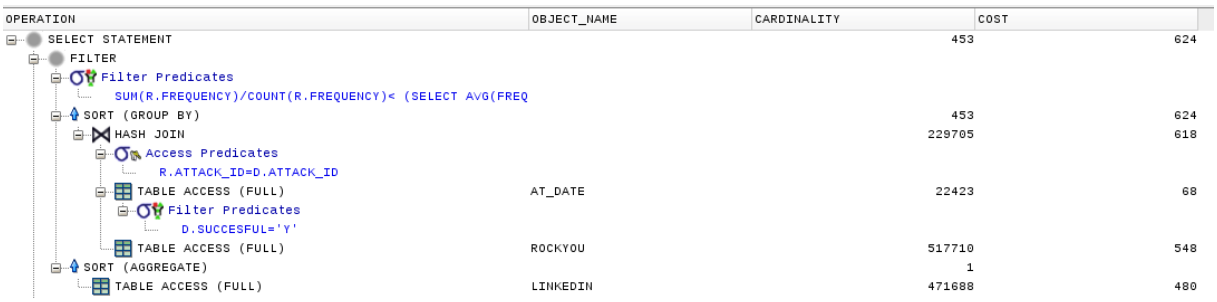
- After implementing indices, the query plan now includes an additional view. This view incorporates a hash join, an access predicate, and leverages two indices.
- Within this new view, values are filtered using the WHERE condition. Row IDs are then connected via a hash join. The WHERE condition is further evaluated through an index range scan with an access predicate, and an index fast full scan is performed on the jobs foreign key.
- The remaining parts of the execution plan are unchanged. While I didn't have a table with millions of rows to test against, the use of indices still managed to reduce the query cost by two units.

WHEN DID THEY MANAGE TO CRACK PASSWORDS THAT ARE RARER THAN AVERAGE?

My second query seeks to identify the specific years during which LinkedIn password breaches occurred with a frequency lower than the overall average breach frequency for LinkedIn passwords. To accomplish this, I linked two tables, with the "Rockyou" table being the more substantial one, containing approximately 520,000 records.

```
select d.YEAR as ev, Round(avg(r.FREQUENCY)) atl
from rockyou r inner join at_date d on r.attack_id=d.attack_id
where d.SUCCESFUL='Y'
group by d.YEAR
having avg(r.FREQUENCY) < (select avg(frequency) from linkedin)
order by d.YEAR;
```

ANALYSIS



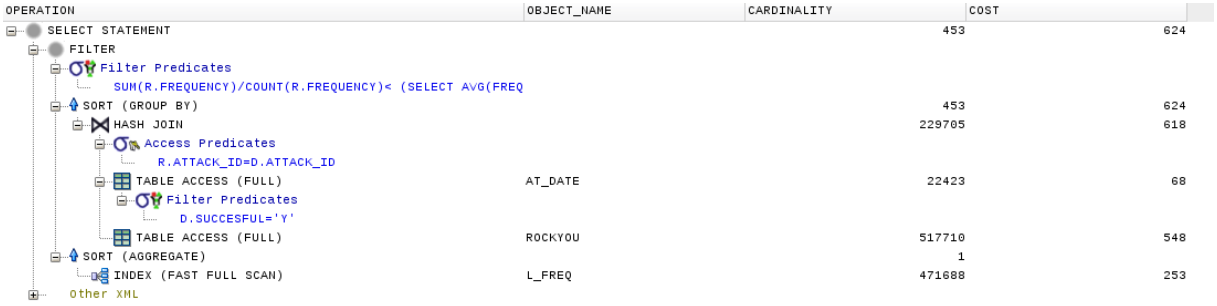
- The execution plan for this query begins by applying filter predicates to calculate the required average value. Subsequently, a GROUP BY operation is performed. Within this grouping, a hash join is executed. This involves a full table scan on the at_date table, where the WHERE condition is applied. Concurrently, a full table scan is performed on the Rockyou table. Finally, the at_date and Rockyou tables are connected using an access predicate within the hash join.

OPTIMIZATION

To optimize the query, I created indexes:

- The attack_id and frequence fields in the Rockyou table,
- and the frequency field in the LinkedIn table.

After optimization, the following implementation plan was proposed by the database manager:



Surprisingly, even after index implementation, this query—which was not particularly costly to begin with—did not see substantial performance improvements, despite joining two large tables..

NOSQL DATABASE MANAGEMENT

For the NoSQL database component, I selected MongoDB due to its document-oriented structure, which aligns well with the nature of the data to be stored.

CONVERT TABLES IN THE DATABASE

To simplify the dataset and optimize the schema for MongoDB, I ultimately consolidated the design to primarily two password-related collections. I prioritized embedding data where collections held inherent significance independently or were frequently accessed together, minimizing the need for joins. For instance, the 'hack type' was embedded within the 'date' collection, as its value is relatively static. While the 'date' collection can function standalone, it is also embedded within the 'passwords' collection due to their frequent co-usage. Similarly, to streamline geographical data, 'country' was embedded within 'city', and 'city' was then embedded within the 'people' collection, as their independent significance was reduced in the NoSQL context. Furthermore, the 'people' collection is embedded within the 'work' collection, given that work-related data, like country or city information, is infrequently modified.

TRANSFER DATA TO MONGODB

Following the design of the collection structures, I proceeded with data importation. I utilized SQL Developer's export functionality to save the relational data in JSON format, which allowed for seamless import into MongoDB. Post-import, I leveraged MongoDB's aggregation framework, specifically the $lookup stage, to transform and assemble the data into the denormalized, embedded structures defined for each collection.

```
itsh@ubuntu:~$ mongoimport --db ic850i --collection country --file ~/Desktop/json/country2.json
2021-05-13T17:16:06.171-0700    connected to: localhost
2021-05-13T17:16:06.172-0700    Failed: error processing document #1: invalid character '"' after object key:value pair
2021-05-13T17:16:06.172-0700    imported 0 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection country --file ~/Desktop/country2.json
2021-05-13T17:28:13.624-0700    connected to: localhost
2021-05-13T17:28:13.658-0700    imported 49 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection city --file ~/Desktop/json/city.json
2021-05-13T17:33:28.044-0700    connected to: localhost
2021-05-13T17:33:28.053-0700    imported 95 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection at_type --file ~/Desktop/json/type.json
2021-05-13T17:34:06.388-0700    connected to: localhost
2021-05-13T17:34:06.392-0700    imported 5 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection at_date --file ~/Desktop/json/date.json
2021-05-13T17:34:24.639-0700    connected to: localhost
2021-05-13T17:34:24.666-0700    imported 3000 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection plus --file ~/Desktop/json/plus.json
2021-05-13T17:35:03.231-0700    connected to: localhost
2021-05-13T17:35:03.306-0700    imported 10000 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection ashleymadison --file ~/Desktop/json/ashley.json
2021-05-13T17:35:44.296-0700    connected to: localhost
2021-05-13T17:35:44.333-0700    imported 4007 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection jobs --file ~/Desktop/json/jobs.json
2021-05-13T17:36:36.823-0700    connected to: localhost
2021-05-13T17:36:36.830-0700    imported 184 documents
itsh@ubuntu:~$ mongoimport --db ic850i --collection people --file ~/Desktop/json/people.json
2021-05-13T17:37:14.666-0700    connected to: localhost
2021-05-13T17:37:14.847-0700    imported 13503 documents
itsh@ubuntu:~$
```

To prevent performance issues and system freezes on my virtual machine, I significantly reduced the datasets for my date table, the Ashley Madison dataset, and the people table. Consequently, the people table now contains a total of 13,000 records, a substantial reduction from its previous size.

myMongo localhost:27017 ic850i

```
db.at_date.aggregate([
{
    $lookup:{
        from: "at_type",
        localField: "type_id",
        foreignField: "type_id",
        as: "types"
        }
},
{
    $out: "at_date"
    }
]);
```

at_date 0.124 sec.

```
Fetched 0 record(s) in 0ms
```

myMongo localhost:27017 ic850i

```
db.ashleymadison.aggregate([
{
    $lookup:{
        from: "at_date",
        localField: "attack_id",
        foreignField: "attack_id",
        as: "dates"
        }
},
{
    $out: "ashleymadison"
    }
]);
```

ashleymadison 2.84 sec.

```
Fetched 0 record(s) in 0ms
```

myMongo localhost:27017 ic850i

```
db.city.aggregate([
{
    $lookup:{
        from: "country",
        localField: "country_id",
        foreignField: "country_id",
        as: "cities"
        }
},
{
    $out: "city"
    }
]);
```

city 0.018 sec.

```
Fetched 0 record(s) in 0ms
```

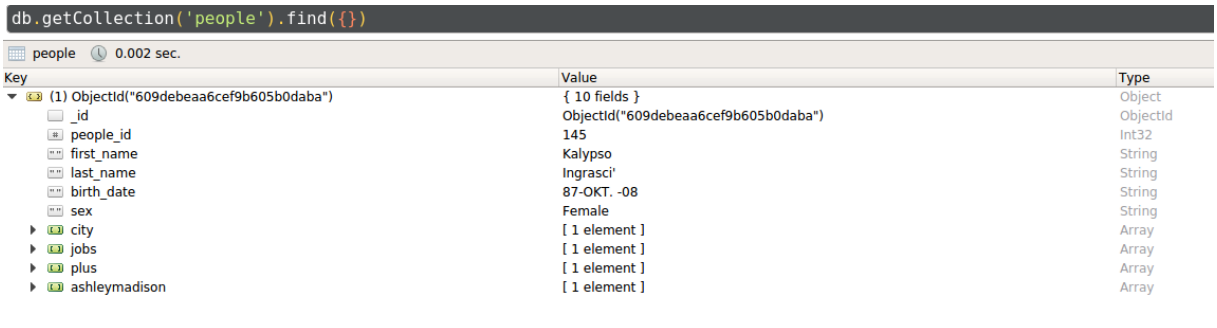
myMongo localhost:27017 ic850i

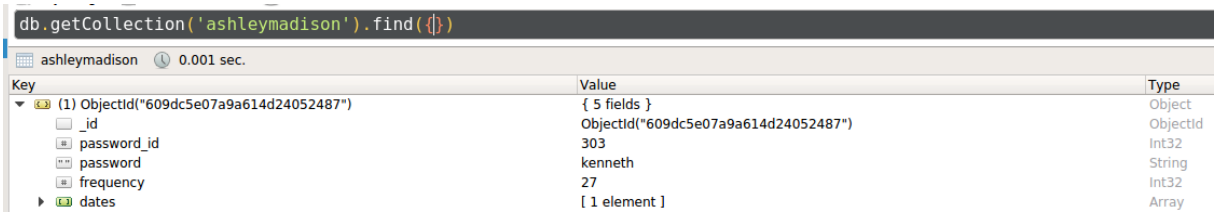
```
db.people.aggregate([
{
    $lookup:{
        from: "city",
        localField: "city_id",
        foreignField: "city_id",
        as: "city"
        }
},
{
    $out: "people"
    }
]);
```

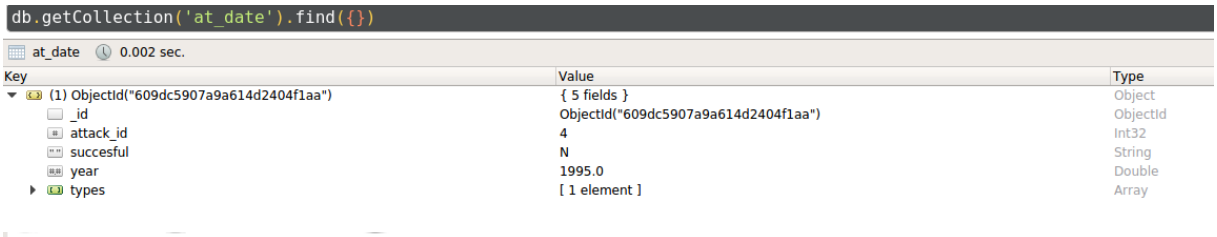
```
db.people.aggregate([
{
    $lookup:{
        from: "plus",
        localField: "plus_id",
        foreignField: "password_id",
        as: "plus"
        }
},
{
    $out: "people"
    }
]);
```

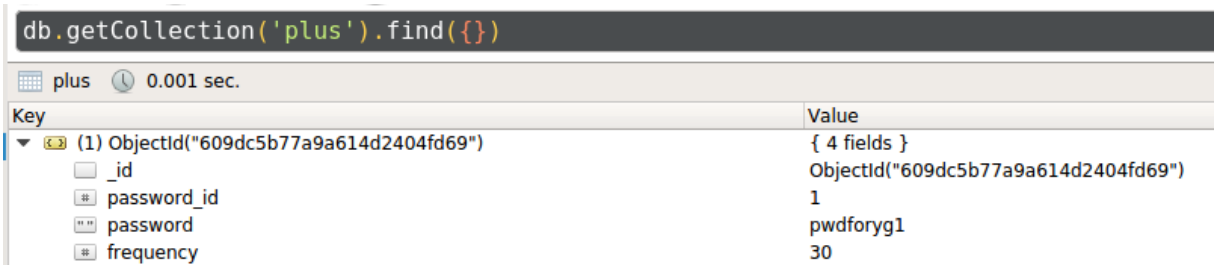
```
db.people.aggregate([
{
    $lookup:{
        from: "ashleymadison",
        localField: "ashley_id",
        foreignField: "ashley_id",
        as: "ashleymadison"
        }
    },
{
    $out: "people"
    }
]);
```

After implementing embedded documents, I removed unnecessary tables and fields from my relational schema. This streamlining ensures that the MongoDB documents reflect the optimized, denormalized structure, enhancing efficiency and relevance.





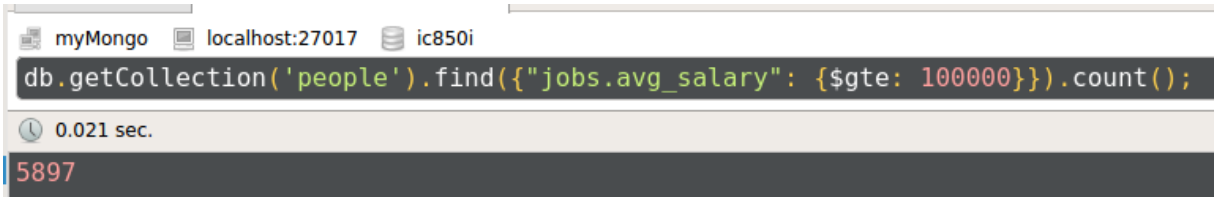




QUERIES

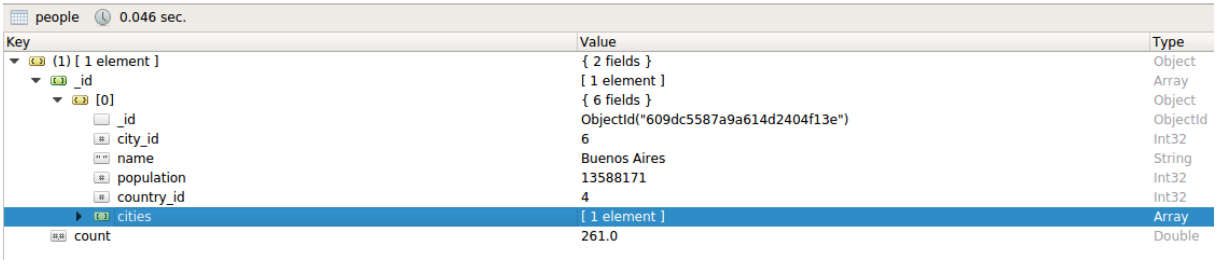
Queries can be found in text format in the attachment.

1. HOW MANY PEOPLE HAVE A JOB ABOVE THE AVERAGE SALARY OF €100,000?



2. WHICH CITY HAS THE MOST EMPLOYEES AND HOW MANY?

```
db.people.aggregate(
[
{$group: {_id:"$city", count: {$sum:1}}},
{$sort: {count: -1}},
{$limit:1}
]
);
```

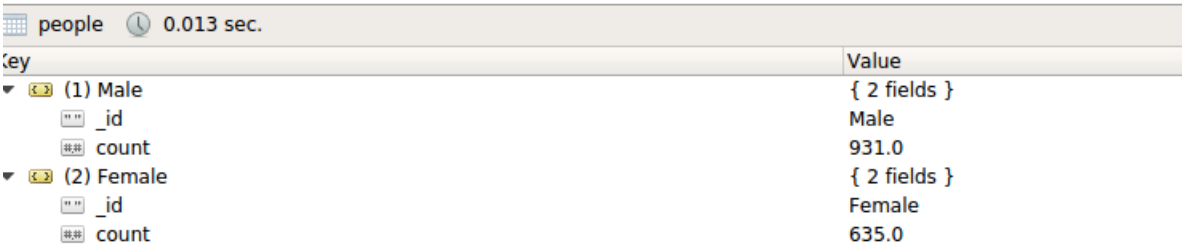


people 0.046 sec.

Key	Value	Type
(1) [1 element]	{ 2 fields }	Object
_id	[1 element]	Array
[0]	{ 6 fields }	Object
_id	ObjectId("609dc5587a9a614d2404f13e")	ObjectId
city_id	6	Int32
name	Buenos Aires	String
population	13588171	Int32
country_id	4	Int32
cities	[1 element]	Array
count	261.0	Double

3. HOW MANY WOMEN AND HOW MANY MEN ARE REGISTERED FROM CITIES OF MORE THAN 10 MILLION PEOPLE?

```
db.people.aggregate(
[
{$match: {"city.population":{$gt:10000000}}},
{$group: {_id:"$sex", count: {$sum: 1}}},
]
);
```



people 0.013 sec.

Key	Value
(1) Male	{ 2 fields }
_id	Male
count	931.0
(2) Female	{ 2 fields }
_id	Female
count	635.0

4. HOW MANY WOMEN HAVE USED PASSWORDS THAT ARE LESS THAN 50% COMMON ON THE DATING SITE?

myMongo localhost:27017 ic850i

```
db.people.aggregate([
{$match: {"sex": "Female", "ashleymadison.frequency":{$lte: 50}}},
{$group:{_id:"$people_id", count:{$sum:1}}}
]);
```

people 0.015 sec.

Key	Value
(1) null	{ 2 fields }
_id	null
count	5694.0

5. HOW MANY PEOPLE ARE REGISTERED IN THE CITY WITH THE SMALLEST POPULATION?

myMongo localhost:27017 ic850i

```
db.people.aggregate([
{$match: {"city.population":{$lt: 60000}}},
{$group:{_id:"$city.city_id",pop:{$min:"$city.population"}, count:{$sum:1}}},
{$sort: {count:-1}},
{$limit:1}
]);
```

people 0.013 sec.

Key	Value
(1) [1 element]	{ 3 fields }
_id	[1 element]
pop	[1 element]
[0]	19383
count	240.0

6. WHAT ARE THE TOP 2 MOST SUCCESSFUL TYPES OF HACKING?

myMongo localhost:27017 ic850i

```
db.at_date.aggregate([
{$match: {succesful: "Y"}},
{$group:{_id:"$types.type_id", count:{$sum:1}}},
{$sort:{count:-1}},
{$limit:2}
]);
```

at_date 0.004 sec.

Key	Value
(1) [1 element]	{ 2 fields }
_id	[1 element]
[0]	3
count	281.0
(2) [1 element]	{ 2 fields }
_id	[1 element]
[0]	1
count	278.0

7. HOW MANY PEOPLE EARN OVER 150,000 BUT USE A PASSWORD ABOVE 65 PERCENT FREQUENCY IN GENERAL?

myMongo localhost:27017 ic850i

```
db.getCollection('people').find({"jobs.avg_salary": {$gte:150000},"plus.frequency": {$gt:65}}).count()
```

0.018 sec.

```
248
```

8. WHICH 4 ARE THE MOST COMMON JOBS AND HOW MANY PEOPLE DO IT?

myMongo localhost:27017 ic850i

```
db.people.aggregate([
{$group:{_id:"$jobs.name", countEmployees:{$sum:1}}},
{$sort:{countEmployees:-1}},
{$limit:4}
]);
```

people 0.021 sec.

Key	Value
(1) [1 element]	{ 2 fields }
_id	[1 element]
[0]	Occupational therapy assistants
countEmployees	204.0
(2) [1 element]	{ 2 fields }
_id	[1 element]
[0]	Electrical transportation equipment installers
countEmployees	198.0
(3) [1 element]	{ 2 fields }
_id	[1 element]
[0]	Stationary engineers
countEmployees	188.0
(4) [1 element]	{ 2 fields }
_id	[1 element]
[0]	Construction inspectors
countEmployees	188.0

9. WHICH 5 PASSWORDS WERE MOST SUCCESSFULLY HACKED?

```
db.ashleymadison.aggregate([
{$match: {"dates.succesful": "Y"}},
{$group:{_id:"$password", countP:{$sum:1}}},
{$sort:{countP:-1}},
{$limit:5}
]);
```

ashleymadison 0.005 sec.

Key	Value
(1) 123456	{ 2 fields }
_id	123456
countP	215.0
(2) password	{ 2 fields }
_id	password
countP	101.0
(3) 12345	{ 2 fields }
_id	12345
countP	100.0
(4) qwerty	{ 2 fields }
_id	qwerty
countP	33.0
(5) 12345678	{ 2 fields }
_id	12345678
countP	31.0

10. WHICH 3 CITIES HAVE THE HIGHEST NUMBER OF PEOPLE EARNING MORE THAN 200,000?

myMongo localhost:27017 ic850i

```
db.people.aggregate([
{$match: {"jobs.avg_salary": {$gte: 350000}}},
{$group:{_id:"$city.name",salary:{$min:"$jobs.avg_salary"}, countP:{$sum:1}}},
{$sort:{countP:-1}},
{$limit:3}
]);
```

people 0.011 sec.

Key	Value
(1) [1 element]	{ 3 fields }
_id	[1 element]
[0]	Delhi
salary	[1 element]
[0]	836610
countP	3.0
(2) [1 element]	{ 3 fields }
_id	[1 element]
[0]	Antwerp
salary	[1 element]
[0]	836610
countP	3.0
(3) [1 element]	{ 3 fields }
_id	[1 element]
[0]	Bratislava
salary	[1 element]
[0]	836610
countP	2.0

ATTACHMENTS

THE SCRIPT THAT CREATES TABLES

```
1.  CREATE TABLE COUNTRY (
2.    country_id NUMBER,
3.    name VARCHAR2(255) NOT NULL,
4.    city_numb NUMBER NOT NULL,
5.    constraint COUNTRY_PK PRIMARY KEY (country_id)
6.  );
7.
8.  CREATE TABLE CITY (
9.    city_id NUMBER,
10.   name VARCHAR2(255) NOT NULL,
11.   population NUMBER NOT NULL,
12.   country_id NUMBER NOT NULL,
13.   constraint CITY_PK PRIMARY KEY (city_id),
14.   CONSTRAINT City_fk FOREIGN KEY (country_id)
15.   REFERENCES Country(country_id)
16.  );
17.
18.  CREATE TABLE At_Type (
19.   type_id NUMBER,
20.   name VARCHAR2(255) NOT NULL,
21.   difficulty VARCHAR2(255) NOT NULL,
22.   constraint TYPE_PK PRIMARY KEY (type_id)
23.  );
24.
25.  CREATE TABLE At_Date (
26.   attack_id NUMBER,
27.   cracked_psw NUMBER,
28.   succesful VARCHAR2(1) NOT NULL,
29.   year DATE NOT NULL,
30.   type_id NUMBER NOT NULL,
31.   constraint DATE_PK PRIMARY KEY (attack_id),
32.   CONSTRAINT Date_fk FOREIGN KEY (type_id)
33.   REFERENCES At_Type(type_id)
34.  );
35.
36.  CREATE TABLE PLUS (
37.   password_id NUMBER,
38.   password VARCHAR2(255) NOT NULL,
39.   frequency NUMBER(2) NOT NULL,
40.   constraint PLUS_PK PRIMARY KEY (password_id)
41.  );
42.
43.  CREATE TABLE LinkedIn (
44.   password_id NUMBER,
45.   password VARCHAR2(255) NOT NULL,
46.   frequency NUMBER(2) NOT NULL,
47.   attack_id NUMBER NOT NULL,
48.   constraint LINKEDIN_PK PRIMARY KEY (password_id),
49.   CONSTRAINT LinkedIn_fk FOREIGN KEY (attack_id)
50.   REFERENCES At_Date(attack_id)
51.  );
52.
```

```
53.  CREATE TABLE AshleyMadison (
54.   password_id NUMBER,
55.   password VARCHAR2(255) NOT NULL,
56.   frequency NUMBER(2) NOT NULL,
57.   attack_id NUMBER NOT NULL,
58.   constraint ASHLEYMADISON_PK PRIMARY KEY (password_id),
59.   CONSTRAINT AshleyMadison_fk FOREIGN KEY (attack_id)
60.   REFERENCES At_Date(attack_id)
61.  );
62.
63.  CREATE TABLE RockYou (
64.   password_id NUMBER,
65.   password VARCHAR2(255) NOT NULL,
66.   frequency NUMBER(2) NOT NULL,
67.   attack_id NUMBER NOT NULL,
68.   constraint ROCKYOU_PK PRIMARY KEY (password_id),
69.   CONSTRAINT RockYou_fk FOREIGN KEY (attack_id)
70.   REFERENCES At_Date(attack_id)
71.  );
72.
73.  CREATE TABLE JOBS (
74.   job_id NUMBER,
75.   name VARCHAR(255) NOT NULL,
76.   avg_salary NUMBER UNIQUE NOT NULL,
77.   difficulty VARCHAR2(255) NOT NULL,
78.   kindof VARCHAR2(5) NOT NULL,
79.   constraint JOBS_PK PRIMARY KEY (job_id));
80.
81.  CREATE TABLE PEOPLE (
82.   people_id NUMBER,
83.   first_name VARCHAR2(50) NOT NULL,
84.   last_name VARCHAR2(50) NOT NULL,
85.   job_id NUMBER NOT NULL,
86.   birth_date DATE NOT NULL,
87.   sex VARCHAR2(255) NOT NULL,
88.   city_id NUMBER NOT NULL,
89.   email VARCHAR2(30) UNIQUE NOT NULL,
90.   rockyou_id NUMBER,
91.   ashley_id NUMBER,
92.   linkedin_id NUMBER,
93.   plus_id NUMBER,
94.   constraint PEOPLE_PK PRIMARY KEY (people_id),
95.   CONSTRAINT People_fk_job FOREIGN KEY (job_id)
96.   REFERENCES Jobs(job_id),
97.   CONSTRAINT People_fk1_city FOREIGN KEY (city_id)
98.   REFERENCES City(city_id),
99.   CONSTRAINT People_fk_rock FOREIGN KEY (rockyou_id)
100.  REFERENCES RockYou(password_id),
101.  CONSTRAINT People_fk_ash FOREIGN KEY (ashley_id)
102.  REFERENCES AshleyMadison(password_id),
103.  CONSTRAINT People_fk_link FOREIGN KEY (linkedin_id)
104.  REFERENCES LinkedIn(password_id),
105.  CONSTRAINT People_fk_p FOREIGN KEY (plus_id)
106.  REFERENCES Plus(password_id)
107. );
```

SEQUENCE SEQUENCE

```
1. CREATE sequence CITY_ID_SEQ
2. START WITH 1
3. INCREMENT BY 1
4. NOCACHE
5. NOCYCLE;
6.
7. CREATE sequence TYPE_ID_SEQ
8. START WITH 1
9. INCREMENT BY 1
10.  NOCACHE
11.  NOCYCLE;
12.
13.  CREATE sequence JOB_ID_SEQ
14.  START WITH 1
15.  INCREMENT BY 1
16.  NOCACHE
17.  NOCYCLE;
18.
19.  CREATE sequence PEOPLE_ID_SEQ
20.  START WITH 1
21.  INCREMENT BY 1
22.  NOCACHE
23. NOCYCLE;
```

PROCEDURE PROCEDURE

```
1. SET SERVEROUPUT ON;
2. CREATE OR REPLACE PROCEDURE AddEmail IS
3. person people%ROWTYPE;
4. sam number;
5. BEGIN
6. FOR person IN (SELECT * FROM people)
7. LOOP
8. select MOD(SYS. DBMS_RANDOM. RANDOM,2) into sam from dual;
9. if sam = 0 THEN
10.  UPDATE people SET email =
11.  first_name||' _'|| last_name||' @gmail.com'
12.  where people.people_id = person.people_id;
13.  else
14.  UPDATE people SET email =
15.  first_name||'.'|| last_name||' @yahoo.com'
16.  where people.people_id = person.people_id;
17.  END IF;
18.  END LOOP;
19.  dbms_output.put_line('Email updated');
20.  END;
21.
22.  execute AddEmail();
```

TRIGGER

```
1. set serverouput on;
2. Create or replace trigger AutoEmail
3. after insert on people
4. for each row
5. Declare
6. email varchar2(255);
7. ev number;
8. begin
9. SELECT EXTRACT(YEAR from birth_date) INTO ev FROM people WHERE
   people_id = :NEW.people_id;
10.  SELECT email INTO email FROM PEOPLE WHERE PEOPLE_ID =
   :NEW.people_id;
11.  if email is null then
12.  insert into people values(people_id_seq.nextval, :NEW.first_name,
   :NEW.last_name, :NEW.job_id, :NEW.birth_date,
13.  :NEW.sex, :NEW.city_id,
14.  (select :NEW.first_name||' _'||:NEW.last_name|| ev||' @gmail.com'
15.  from people
16.  where people.people_id = :NEW.people_id),:NEW.rockyou_id,
   :NEW.ashley_id, :NEW.linkedin_id, :NEW.plus_id);
17.  dbms_output.put_line('Automatic email generated.');
18.  else
19.  dbms_output.put_line('Email was filled.');
20.  end if;
21.  end;
```

OPTIMIZATION 1:

```
1. select COUNT(p.PEOPLE_ID), c.NAME
2. from people p inner join city c on p.city_id=c.city_id
3. inner join jobs j on p.job_id=j.job_id
4. where j.AVG_SALARY>80000
5. and (EXTRACT(YEAR FROM sysdate)-EXTRACT(YEAR FROM p.BIRTH_DATE)))<25
6. group by c.NAME;
```

OPTIMIZATION 2

```
1. select d.YEAR as ev, Round(avg(r.FREQUENCY)) atl
2. from rockyou r inner join at_date d on r.attack_id=d.attack_id
3. where d.SUCCESFUL='Y'
4. group by d.YEAR
5. having avg(r.FREQUENCY) < (select avg(frequency) from linkedin)
6. order by d.YEAR;
1. Query
2.
3. db.getCollection('people').find({"jobs.avg_salary": {$gte:
   100000}}}count();
```

2. QUERY

```
1. db.people.aggregate(
2. [
1. {$group: {_id:"$city", count: {$sum:1}}}},
2. {$sort: {count: -1}},
3. {$limit:1}
4. ]
5. );
```

3. QUERY

```
1. db.people.aggregate(
2. [
3. {$match: {"city.population":{$gt:10000000}}}}
4. {$group: {_id:"$sex", count: {$sum: 1}}},
5. ]
6. );
```

4. QUERY

```
1. db.people.aggregate([
2. {$match: {"sex": "Female", "ashleymadison.frequency":{$lte: 50}}},
3. {$group:{_id:"$ashleymadison.password_id", count:{$sum:1}}}
4. ]);
```

5. QUERY

```
5. db.people.aggregate([
6. {$match: {"city.population":{$lt: 60000}}}}
7. {$group:{_id:"$city.city_id",pop:{$min:"$city.population"},
   count:{$sum:1}}},
8. {$sort: {count:-1}},
9. {$limit:1}
10.  ]);
```

6. QUERY

```
11.  db.at_date.aggregate([
12.  {$match: {succesful: "Y"}},
13.  {$group:{_id:"$types.type_id", count:{$sum:1}}},
14.  {$sort:{count:-1}},
15.  {$limit:2}
16.  ]);
```

7. QUERY

```
17.  db.getCollection('people').find({"jobs.avg_salary":
   {$gte:150000},"plus.frequency": {$gt:65}}).count()
```

8. QUERY

```
db.people.aggregate([
{$group:{_id:"$jobs.name", countEmployees:{$sum:1}}},
{$sort:{countEmployees:-1}},
{$limit:4}
]);
```

9. QUERY

```
db.ashleymadison.aggregate([
{$match: {"dates.succesful": "Y"}},
{$group:{_id:"$password", countP:{$sum:1}}},
{$sort:{countP:-1}},
{$limit:5}
]);
```

10. QUERY

```
db.people.aggregate([
{$match: {"jobs.avg_salary": {$gte: 350000}}}
{$group:{_id:"$city.name",
salary:{$min:"$jobs.avg_salary"}, countP:{$sum:1}}}},
{$sort:{countP:-1}},
{$limit:3}
]);
```