**[Data Analysis] Database Module B - project**

**Online Learning Platform**

**540438 - Maurizio La Rosa**

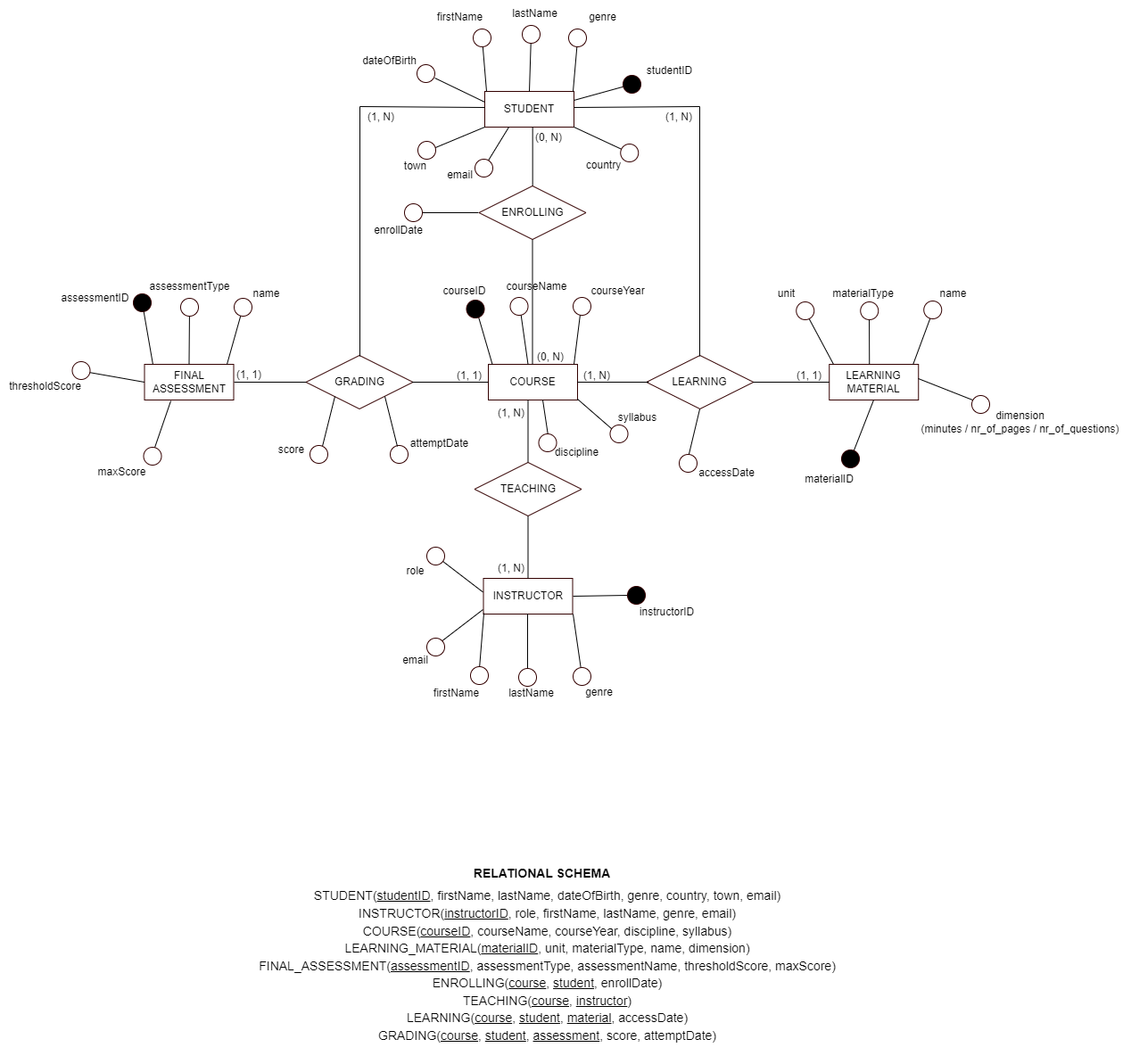
**Presenting the project**

The project consists in the comparison of the behaviour of the DBMSs studied ([MySQL](https://www.mysql.com/it/), [MongoDB](https://www.mongodb.com), [Cassandra](https://cassandra.apache.org/_/index.html), [Redis](https://redis.io), and [Neo4j](https://neo4j.com)) when performing queries on the same data. The dataset must reproduce the data of an online learning platform characterized by the presence of students enrolled to courses and accessing the courses' learning materials. A fake dataset has been implemented via programming language (Python) and four csv files of increasing size (in terms of rows) have been created (the csv files have 250000, 500000, 750000 and 1000000 rows, respectively). Each dataset must be fed to all of the cited DBMSs and four queries of increasing complexity (from the point of view of the number of entities involved and the selection filters) must be performed on each dataset within each DBMS. The queries must be implemented by interaction between a programming language and each DBMS. Each DBMS implements a set of drivers for the most common programming languages and references are available in the DBMS web pages. I opt for interacting through *Python*. Finally, query execution times must be recorded at millisecond precision, stored into a spreadsheet and plotted by means of bar plots.

**Database schema hypothesis**

In order to simplify the construction of the simulated data, I built a conceptual schema of the database. The aim is to simulate data that reproduce the schema. The data are created programmatically, either by using the *faker* Python module either by exploiting the *random* Python module. Data are stored into dictionaries of lists. Each dictionary represents a table (entity or relationship), where the key is a table attribute and each list of values is the attribute’s instances.

The schema, reproduced below, incorporates 5 entities and 4 relationships. The concept is somewhat a simplification of the way I consider the full actual schema of this kind of database (learning platform). The relationships *grading* and *learning* link three entities each (*grading*: *student*, *course* and *assessment*; *learning*: *student*, *course* and *learning material*). It would probably be better if they were replaced by two distinct relationships: instead of only *grading* we could have one relationship linking *course* and *assessment* and another one linking *student* and *assessment*. This way we could have a full representation of each course and its assessment.



**course - final assessment**

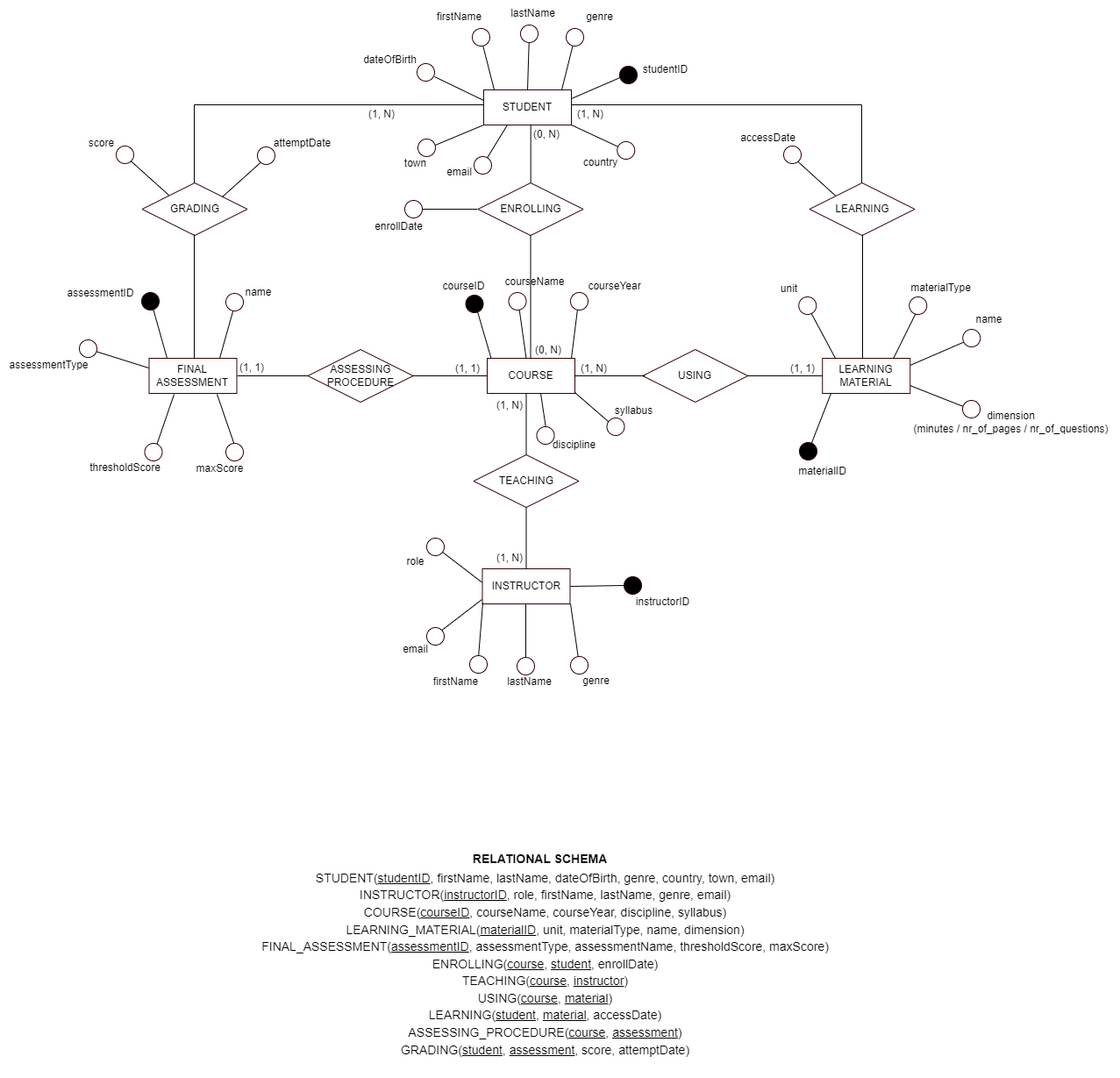
The way the concept is expressed at the moment, *assessment* is present in the *grading* relationship only if a student has actually undergone assessment in a given date, and there is no other relationship involving *course* and *assessment*. So, if a student enrolled in a course has not undergone the course final assessment, the assessment data cannot be linked to the course data. Notice that the full assessment data is present in the assessment table, but the can be linked to a course only through a relationship, but the only one is *grading* which stores data only for actual assessments undergone by students, not potential assessments.

**course - learning material**

The same reasoning applies to the *learning* relationship. It stores connection of courses and learning materials only if students have actually accessed the learning material. Though the full learning material data are stored in the *learning material* table, they cannot be linked to courses unless they are accessed by students and then stored in the *learning* table.

**Full database schema**

The full schema could be represented as follows, with eleven constructs instead of nine (5 entities and 6 relationships):



**course - learning material**

*course* is linked to *learning material* through a specific relationship (*using*), while *student* is linked to *learning material* through the *learning* relationship. While the *using* relationship between *course* and *learning material* stores the full set of links between a course and the learning material used within the course, the *learning* relationship stores only learning material that students have accessed to. Also, while in the *using* relationship learning materials are uniquely identified, in the *learning* relationship they might appear more than once because students can access multiple times the same learning material or, better, more students can access the same learning material.

**course - final assessment**

The same is true for the new relationship between *course* and *final assessment*, *assessing procedure*. It stores the full set of links between a course and its final assessment, while the *grading* relationship only stores assessments actually undergone by students. Again, while the *assessing procedure* uniquely identifies final assessments (one assessment per course, each course has a different assessment), in the *grading* relationship the same assessment might appear multiple times: different students enrolled to the same course might have taken the assessment or the same student might undergo the course assessment multiple times if she hasn't succeeded the first time.

Anyway, for the purpose of the project it is not necessary, in my view, to build the full schema. The first schema hypothesis can be seen as a simplification of the full schema allowing to perform queries complex enough for the project assignment demands. The range of queries is more limited than that of the full schema: the full schema would allow recovering all data connected to each course of the platform (assignments and learning materials), but, all in all, it is more interesting to know which material has been accessed by a student, or if a student has performed the final assignment of a course, and these kinds of queries (involving three entities) are possible with this schema.

**Final dataset (csv file)**

The final output of the database building procedure (stored in the FakeDB folder), according to the project guidelines, must be a csv file of 1 million records to be imported into each of the indicated DBMSs. I don't need to import every entity or relationship in my relational schema, it is sufficient to import a unique csv file storing enough information. I can use the *learning* relationship in my relational schema to gather information on students, on courses and on the learning materials. These information (attributes) can all be stored into a csv file which will have a number of records equivalent to those in the *learning* table (more than 1 million) and as many columns (attributes) as those present in the *student* table, in the *course* table and in the *learning material* table (plus the unique attribute in the *learning* table). After importing the csv into the datasets, I can run queries based on the attributes of the entities involved (students' gender or nationality, courses' discipline, types of learning materials accessed, etc.).

**Importing csv files into the DBMSs**

The step that must precede the establishment of an interaction between Python and the DBMSs, is the import of the four csv files that must be used to perform the query execution tests. Each DBMS has its import utilities, and I present them individually. I generally prefer to perform import at the DBMS level rather than accessing the database from the programming language interface and copying the data into the database (for example by looping over the data structure and inserting data rows into a MySQL table in the database). However, in some cases, importing via programming language interaction has been judged the best strategy (for Redis, for example).  
The import phase is generally performed through DBMS defined routines, however, not via the application programming language.

After introducing the procedure required for the import phase, I present the driver for the Python - DBMS interaction. I do this for each DBMS. Generally, I explain hor to connect to a DBMS and authenticate, how to choose a database and perform queries. All the previous methodologies are introduced via a specific notebook for each DBMS. Finally, I implement the four queries for each of the four differently sized datasets and present the response time. Query executions and execution times storage is performed via a second dedicated notebook (one for each DBMS).

**Defining the queries**

Here I will define the four queries that must be performed for each database instance on each of the five DBMSs of interest.

**Query 1**

Show the first and last names of the students enrolled to the course with courseID number *192*

**Query 2**

Show the names of the courses belonging to the discipline *statistics* for year *2022*

**Query 3**

Show the number of learning materials of type 'slides' accessed by the students enrolled to courses belonging to the discipline *maths* who have *gmail.com* domain in their email

**Query 4**

Show the first and last names and the country of the students enrolled to the courses belonging to the discipline *psychology* who come from any of the two states of *Korea* and were born before year *2000*, sort them by surname

**MySQL - Methodologies applied for loading the data and performing the queries**

**Preliminary operations: import csv files into MySQL (LOAD DATA statement)**

#### **Core syntax**

Import of a csv file in MySQL can be performed by a LOAD DATA statement. The file to be imported must be specified after the INFILE keyword, while the table name where the data will be copied must be specified after the INTO TABLE keywords. So, the basic structure of a LOAD DATA statement is as follows:

LOAD DATA

INFILE ‘mycsvfile.csv’

INTO TABLE mytable;

#### **Optional clauses**

- FIELDS and LINES

It is also useful to include some optional keywords in the statement, providing the DBMS information on how the data is organised into the csv file. These may refer to FIELDS (TERMINATED BY and ENCLOSED BY) or LINES (TERMINATED BY) and are used to tell the DBMS, in order:

* how fields (columns, attributes) are separated by one another (by commas ‘,’ in the following case);
* that characters within double quotes must not be considered as separators (‘“’). Since the data contained in each field are not necessarily enclosed within double quotes (they usually are when there is a comma that does not separate fields), the keyword OPTIONALLY is also used;
* to perform a new insertion (insert a new row) when meeting the special character indicating a new line (‘\n’):

LOAD DATA

INFILE ‘mycsvfile.csv’

INTO TABLE mytable

FIELDS TERMINATED BY ‘,’

OPTIONALLY ENCLOSED BY ‘“’

LINES TERMINATED BY ‘\n’;

- IGNORE

By specifying to IGNORE 1 ROWS we declare that the first row of the csv file must not be inserted into the table (we imply, usually, that it contains the attribute names of the table).

- LOCAL

Finally, it is to be considered that ‘mycsvfile.csv’ is expected to be located on the server host. If it is not the case (we want to load a text file located on the client host) the keyword LOCAL must be specified immediately after LOAD DATA. The complete statement to import a csv file into MySQL is then as follows:

LOAD DATA

LOCAL INFILE ‘mycsvfile.csv’

INTO TABLE mytable

FIELDS TERMINATED BY ‘,’

OPTIONALLY ENCLOSED BY ‘“’

LINES TERMINATED BY ‘\n’

IGNORE 1 ROWS;

#### **A note on the use of LOCAL:**

Since there are potential security implications within the process of transferring a file from the client to the server (these are specified here and consist either in the theoretical possibility for a server to gain access to any part of the file system to which the client has access or in the possibility, in a web environment, that a user connected to the web server is allowed to read any files that the web server has read access to), the use of LOCAL may return an error if the client host and the server host are not configured appropriately.For example, if I run MySQL from a Docker container, select a database and create a table with a schema matching the header of the csv file, I can run and perform the afore mentioned LOAD DATA statement, and be returned the error:

ERROR 3948 (42000): Loading local data is disabled; this must be enabled on both the client and server sides.

For control over local data loading, MySQL permits this capability to be enabled or disabled.

At the server level, the capability is controlled by the global system variable local\_infile. So, to check if it is enabled at the server level, a SHOW GLOBAL VARIABLES statement can be performed:

SHOW GLOBAL VARIABLES LIKE ‘local\_infile’;

This is usually set to OFF, so a SET GLOBAL statement on the aforementioned variable would allow us to change it to ON:

SET GLOBAL local\_infile = True;

At the client level, it is sufficient to start MySQL with the option local\_infile=1 to enable the capability:

mysql –local\_infile=1 -uroot -p

#### **Notes when running MySQL from a Docker container:**

A couple of further notes are required when running MySQL within a Docker container. Since a Docker container has its own file system, MySQL cannot access parts of the file system not mounted to the container. In particular, the csv file cannot be loaded by MySQL from outside the container. Hence it has to be copied to the container with the docker cp command. This command allows either to copy a file from the container to the local machine or the reverse (which is what we want here). We need to simply pass the source path and file name and the destination path. In our case it would be:

docker cp path/mycsvfile.csv container:/path

where container is the container name. To put the file in the root folder of the container one can simply type *container:/* as destination.

Furthermore, I want to load the csv file from within a SQL script. This script is organised in a CREATE TABLE statement and a LOAD DATA statement (it is advised to also indicate the database where to create the table, or to create it as well). The table creation specifies a schema that matches exactly the first row of the csv file, which contains the table header. The import statement specifies the csv file name and it optionally indicates its path. Actually, I copied the csv files (the four differently sized files) directly into the container, without specifying any container subfolder, hence I would simply indicate the file name in single quotes after the LOCAL INFILE clause of the LOAD DATA statement. Exactly as it is shown in the above templates.

A final consideration regards the execution of an external script from a MySQL instance run from within a container. As the [MySQL Docker Hub](https://hub.docker.com/_/mysql) web page points out under the paragraph Restoring data from dump files, MySQL must be executed from the container together with the indication of the script that we want to be executed:

docker exec -i container mysql –local\_infile=1

-uroot -p[PASSWORD] < path/myscript.sql

Notice the use of the option local\_infile=1 when starting the server to enable file loading at the client level. Previously, after starting the MySQL container, we must have set the local\_infile global variable on as previously specified. Also notice that this procedure does not allow us to input the password interactively. It must be provided when running the docker exec command from the command line.

After successful execution of the script we are returned to the system folder from which we have run the docker exec command. By entering MySQL again we can check if the scripted actions have been actually performed on the MySQL server.

**Performing the queries and storing the queries execution time**

#### **Python - MySQL interaction**

Prior to performing the queries we import the required modules (the MySQL connector and the time and csv modules) and establish a connection with the MySQL instance running in Docker.

# import modules  
import mysql.connector as connector # MySQL driver  
import time # time-related functions to register query execution times  
import csv # read and write csv files  
  
# create connection object  
conn = connector.connect(host = '127.0.0.1', port = '3306', user = 'root', password = 'X4mPpd3V', database = 'dbB\_MYSQL\_test')  
  
# create cursor object  
cursor = conn.cursor()

#### **Query the datasets**

I create a dictionary of lists for each of the four datasets. In these dictionaries the keys are the query names and the values are the 31 query execution times: in fact I attach the value of the query execution time of the most recent query to the list. Since query execution times are required in milliseconds, prior to attaching them, I multiply them by 1000 and round them to the fifth decimal precision. The above summarized actions (for each of the four queries on each of the four datasets) are performed by following a standard succession of steps. Each step is encapsulated within a notebook cell (so each query is performed 31 times by using three notebook cells), as follows:

* step 1: define the query, perform it for the first time, contextually create timestamps prior and after query execution, print query result;
* step 2: compute execution time of the first query execution and store it within the corresponding dictionary list;
* step 3: [thirty times] perform query execution while creating prior and following timestamps, compute execution time and store it within the corresponding dictionary list, reset the cursor to allow repeating the query.

For each dataset, after having performed the four queries, I will finally compute the mean of the query executions from step 3. Together with the first query execution, this mean value will be stored into a new dictionary, specific to a dataset. Originally, I would use these four new dictionaries to save the query execution times into a csv file for constructing histograms. I later resolved to save all the 31 recorded query execution times and pass them all to Microsoft© Excel to process them.

smallDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
mediumDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
largeDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
humongousDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}

# mean function (used to compute mean execution time of the 30 grouped queries)  
def mean(aList):  
 n = len(aList)  
 sum = 0  
 for value in aList:  
 sum += value  
 return sum / n

#### **Dataset with 250k records**

I start with the smallest dataset.

**Query1**

# step 1  
small\_sql1 = 'SELECT DISTINCT firstName AS name, lastName AS surname FROM smallDB AS S WHERE courseID = 192'  
  
before = time.time()  
cursor.execute(small\_sql1)  
after = time.time()  
  
small\_query1 = cursor.fetchall()  
for name, surname in small\_query1:  
 print(name, surname)

Custodia Hidalgo  
Sarah Lara  
Narciso Ferrán  
Patrícia Leite  
Vigilija Gaižauskas  
Casandra Arenas  
Ledün Soylu  
Arthur Laroche  
Ana Narušis  
Nath Nicolas  
Émile Nicolas  
Cathrine Lie  
Ingeborg Amundsen

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query1'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 cursor.execute(small\_sql1)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query1'].append(round(msec\_duration, 5))  
 cursor.reset()

**Query2**

# step 1  
small\_sql2 = 'SELECT DISTINCT courseName AS name FROM smallDB WHERE discipline = \'statistics\' AND courseYear = 2022'  
  
before = time.time()  
cursor.execute(small\_sql2)  
after = time.time()  
  
small\_query2 = cursor.fetchall()  
for course in small\_query2:  
 print(course[0])

Econometrics: Methods and Applications  
Exploratory Data Analysis  
Understanding Clinical Research: Behind the Statistics  
Introduction to Probability and Data with R  
Bayesian Statistics: From Concept to Data Analysis  
Introduction to Statistics  
Python and Statistics for Financial Analysis  
Basic Statistics  
Foundations: Data, Data, Everywhere

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query2'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 cursor.execute(small\_sql2)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query2'].append(round(msec\_duration, 5))  
 cursor.reset()

**Query3**

# step 1  
small\_sql3 = 'SELECT COUNT(materialID) FROM smallDB WHERE materialType = \'lecture slides\' AND discipline = \'maths\' AND email LIKE \'%gmail.com\''  
  
before = time.time()  
cursor.execute(small\_sql3)  
after = time.time()  
  
small\_query3 = cursor.fetchall()  
for count in small\_query3[0]:  
 print(count)

838

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query3'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 cursor.execute(small\_sql3)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query3'].append(round(msec\_duration, 5))  
 cursor.reset()

**Query4**

# step 1  
small\_sql4 = 'SELECT DISTINCT firstName AS name, lastName as surname, country FROM smallDB WHERE discipline = \'psychology\' AND country LIKE \'%orea\' AND dateOfBirth LIKE \'1%\' AND courseYear = 2023 ORDER BY surname ASC;'  
  
before = time.time()  
cursor.execute(small\_sql4)  
after = time.time()  
  
small\_query4 = cursor.fetchall()  
for name, surname, country in small\_query4:  
 print(name, surname, 'from', country)

Cathrine Lie from South Korea  
Lynda Reynolds from Korea  
Raghav Sura from North Korea

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query4'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 cursor.execute(small\_sql4)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query4'].append(round(msec\_duration, 5))  
 cursor.reset()

I store the execution time of the first query execution and of the mean of the following 30 query executions into a new dictionary.

smallDataset = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
for key in smallDict:  
 smallDataset[key].append(smallDict[key][0])  
 mean30 = mean(smallDict[key][1 : 31])  
 smallDataset[key].append(round(mean30, 5))  
smallDataset

{'query1': [3391.55889, 167.68118],  
 'query2': [248.03996, 206.50523],  
 'query3': [243.83307, 213.16119],  
 'query4': [259.29236, 268.66496]}

The execution of the queries on the progressively larger datasets is omitted to avoid unnecessarily increasing the report length.

[…]

**Cassandra - Methodologies applied for loading the data and performing the queries**

**Preliminary operations: import csv files into Cassandra (COPY FROM)**

A csv file can be imported into a Cassandra table via the COPY FROM CQL [command](https://cassandra.apache.org/doc/4.1/cassandra/tools/cqlsh.html#copy-from), although for large files it is suggested, to rely on [bulk loading](https://cassandra.apache.org/doc/latest/cassandra/operating/bulk_loading.html), an operation that can be performed either with the sstableloader command or with the nodetool import command. Neither of the two is a DML command and they need being executed from the Cassandra shell.A major limitation of these tools resides in the fact that they can import external files in the form of sstables, not directly csv files. Hence, bulk-loading external data that is not in sstable form, should first be stored into sstables, a process that requires using the CQLSSTableWriter Java class. After experimenting with a file with a couple of thousands of rows, I have resolved to rely on the COPY FROM CQL command.

#### **Syntax**

To use the COPY FROM command we must first select the keyspace (USE mykeyspace) of the table that we want the csv data to be imported into. The table must exist within the selected keyspace and its schema must replicate the header of the csv file and related data types. COPY mytable FROM ‘mycsvfile.csv’; Equivalently, without the need to first select the keyspace, we could specify the table as a keyspace attribute: COPY mykeyspace.mytable FROM ‘mycsvfile.csv’; It is to be considered that when we perform a CREATE TABLE CQL command there is no guarantee that the output table will keep the inputted column order. So, although in table creation we might have specified the same column order of the first row of the csv (header), when importing it is advisable to specify the column list (within round brackets after the table name) in the same order as it appears in the csv header.

COPY tablename(column1, column2, …) FROM ‘mycsvfile.csv’;

#### **Options**

Options are specified after the keyword WITH and separated by the keyword AND. Options are useful to change the CSV format. In fact, by default, Cassandra expects the CSV data to consist of fields separated by commas (,), records separated by line separators (a newline, ), field values enclosed in double-quotation marks (““) and a header row not to be present.

- HEADER

If the first row of the csv file contains the column names, we must specify HEADER = true after the filename, so that it is ignored when importing.

COPY tablename(column1, column2, …)

FROM ‘mycsvfile.csv’

WITH HEADER = true;

- DELIMITER

To avoid problems in column identification, it is also advisable to specify the character used within the csv file to separate columns.

COPY tablename(column1, column2, …) F

ROM ‘mycsvfile.csv’

WITH HEADER = true

AND DELIMITER = ‘,’;

- QUOTE

If there are commas within column values, column values usually come written within double quotes. If a different enclosing characyer is used, it must be specified with the QUOTE option.

COPY tablename(column1, column2, …)

FROM ‘mycsvfile.csv’

WITH HEADER = true

AND DELIMITER = ‘,’

AND QUOTE = ‘“’;

However, in my datasets, usually I don’t use double quotation to enclose field values, so to preserve data types. Hence, only strings that contain commas are enclosed in double quotes. This may cause COPY FROM to experience troubles because, if QUOTE is used, not every field value comes with this kind of enclosure, while, if QUOTE is not used,COPY FROM will likely fail in identifying the correct number of field when reaching a row where some values are double quoted. This is likely to happen for the courseName field: course names may include commas. To increase compatibility, it is then advisable to double quote dictionary values when creating the data with Python and leave to the DBMSs the task of identifying the correct data type.

#### **Notes when running Cassandra from a Docker container**

In loading the csv file from the local machine, we must take into account the usual feature of Docker virtual environments. They come with a file system of their own, so a DBMS run from within a container has access to this file system, not to the local machine file system. Hence, the csv file must be imported into the container via the usual docker cp command.

docker cp path/mycsvfile.csv container:/path

Contrary to what has been specified in the MySQL and MongoDB cases, I run the COPY FROM command from cqlsh, not from a system folder or from a container shell. Also, the file has been copied in the root folder of the container file system, hence I don’t need to include a path for the csv file, I just specify its name within single quotes.

**Performing the queries and storing the queries execution time**

#### **Python - Cassandra interaction**

Prior to performing the queries we import the required modules (the Cassandra Python driver and the time and csv modules), establish a connection with the Cassandra instance running in Docker and choose the keyspace on which we will perform the queries.

from cassandra.cluster import Cluster # MySQL driver  
import time # time-related functions to register query execution times  
import csv # read and write csv files  
  
# instantiate a cluster  
cluster = Cluster(['127.0.0.1'])  
  
# create a session by connecting to the cluster  
session = cluster.connect()  
  
# associate a keyspace to the session  
session.set\_keyspace('dbb\_cassandra\_test')

#### **Query the datasets / keyspaces**

I create a dictionary of lists for each of the four keyspaces. In these dictionaries the keys are the query names and the values are the 31 query execution times: in fact I attach the value of the query execution time of the most recent query to the list. Since query execution times are required in milliseconds, prior to attaching them, I multiply them by 1000 and round them to the fifth decimal precision. The above summarized actions (for each of the four queries on each of the four keyspaces) are performed by following a standard succession of steps. Each step is encapsulated within a notebook cell (so each query is performed 31 times by using three notebook cells), as follows:

* step 1: create index if needed, define the query, perform it for the first time, contextually create timestamps prior and after query execution, print query result;
* step 2: compute execution time of the first query execution and store it within the corresponding dictionary list;
* step 3: [thirty times] perform query execution while creating prior and following timestamps, compute execution time and store it within the corresponding dictionary list , reset the cursor to allow repeating the query.

For each dataset, after having performed the four queries, I will finally compute the mean of the query executions from step 3. Together with the first query execution, this mean value will be stored into a new dictionary, specific to a dataset. Originally, I would use these four new dictionaries to save the query execution times into a csv file for constructing histograms. I later resolved to save all the 31 recorded query execution times and pass them all to Microsoft© Excel to process them.

smallDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
mediumDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
largeDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
humongousDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}

# mean function (used to compute mean execution time of the 30 grouped queries)  
def mean(aList):  
 n = len(aList)  
 sum = 0  
 for value in aList:  
 sum += value  
 return sum / n

#### **Table with 250k records**

I start with the smallest table.

**Query1**

It is to be noted that, in relation to its focus on performance, Cassandra does not allow the use of the DISTINCT keyword on columns that are not partition keys (primary keys). In Query 1, however, we are only interested in the names of the students enrolled to the course having ID = 192, there is no point in having a student’s name repeated more than once. Each student enrolled to course 192, however, has accessed to various learning materials of course 192, hence the query result, without the use of DISTINCT will be a bag, rather than a set. Duplicate values must be handled in some way, in order to display each student only once. I have considered two possibilities to reach the desired result:

* the first one uses CQLSH and an external csv file. It requires, within the selected keyspace, creating a new table with three fields (student first name, student last name and course ID) and setting the first two as primary key. Then the same three fields are to be copied from the entire original table to a csv file and copied back from the csv file to the table having firstname and lastName as primary key. Then the query on the new table can be run by using the DISTINCT option:
* ‘CREATE TABLE query1\_temp (
  + firstName VARCHAR, lastName VARCHAR, courseID VARCHAR, PRIMARY KEY(firstName, lastName));’

‘COPY smallDB(firstName, lastName, courseID) TO ’path/query1\_temp.csv’

WITH HEADER = TRUE AND DELIMITER = ‘,’;’

‘COPY query1\_temp(firstname, lastName, courseID) FROM

’path/query1\_temp.csv’ WITH HEADER = TRUE AND DELIMITER = ‘,’;’ ‘SELECT DISTINCT firstName, lastName FROM query1\_temp WHERE courseID =

192;’

* the second one just considers performing the query on the original table without the use of the DISTINCT keyword. The query result is then processed via programming language to obtain the unique values of the students enrolled to course 192. Python is suitable for this purpose, having in store set objects than do not allow element replicas.

Both methods will affect the query execution time, if all the steps are to be taken into account. The second method seems to me the cleanest one and I will apply it for Query 1. In particular, I will add to step 1 a new substep: prior to recording timestamps I create a set (resSet) where I want to store unique values from the query result. After having defined thid object and created the index required by the query: I record the first timestamp, I run the query, I manipulate the query result by storing the rows into the resSet object, I record the last timestamp. Then the query result can be displayed.

# step 1  
resSet1 = set()  
session.execute('CREATE INDEX IF NOT EXISTS query1Index ON smalldb(courseid);')  
small\_cassandra1 = 'SELECT firstName, lastName FROM smalldb WHERE courseid = \'192\';'  
  
before = time.time()  
small\_query1 = session.execute(small\_cassandra1)  
for row in small\_query1:  
 resSet1.add(row)  
after = time.time()  
  
for element in resSet1:  
 print(element[0], element[1])

Casandra Arenas  
Ledün Soylu  
Cathrine Lie  
Custodia Hidalgo  
Patrícia Leite  
Arthur Laroche  
Narciso Ferrán  
Ingeborg Amundsen  
Sarah Lara  
Vigilija Gaižauskas  
Nath Nicolas  
Ana Narušis  
Émile Nicolas

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query1'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 resSet1 = set()  
 before = time.time()  
 small\_query1 = session.execute(small\_cassandra1)  
 for row in small\_query1:  
 resSet1.add(row)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query1'].append(round(msec\_duration, 5))

**Query 2**

Query 2 requires selection on more than one field value (discipline must be ‘statistics’ and the year of the course must be ‘2022’. In this case two indices must be created, but Cassandra requires the ALLOW FILTERING clause because double indexing may negatively impact query performance. Even in this case we need unique values, hence I add results to a Python set after query completion.

# step 1  
resSet2 = set()  
session.execute('CREATE INDEX IF NOT EXISTS query2Index1 ON smalldb(discipline);')  
session.execute('CREATE INDEX IF NOT EXISTS query2Index2 ON smalldb(courseyear);')  
small\_cassandra2 = 'SELECT coursename FROM smalldb WHERE discipline = \'statistics\' AND courseyear = \'2022\' ALLOW FILTERING;'  
  
before = time.time()  
small\_query2 = session.execute(small\_cassandra2)  
for row in small\_query2:  
 resSet2.add(row)  
after = time.time()  
  
for element in resSet2:  
 print(element[0])

Basic Statistics  
Exploratory Data Analysis  
Bayesian Statistics: From Concept to Data Analysis  
Python and Statistics for Financial Analysis  
Foundations: Data, Data, Everywhere  
Understanding Clinical Research: Behind the Statistics  
Econometrics: Methods and Applications  
Introduction to Probability and Data with R  
Introduction to Statistics

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query2'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 small\_query2 = session.execute(small\_cassandra2)  
 for row in small\_query2:  
 resSet2.add(row)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query2'].append(round(msec\_duration, 5))

**Query 3**

In this case the hardest difficulty was in trying to implement an index that could behave similarly to the MySQL LIKE, by matching patterns in string. In Cassandra custom indices can be created, in particular the so-called SASI indexes which can be set on three different modes: PREFIX (default), CONTAINS or SPARSE. The first one allows use of a syntax such as the following:

SELECT fieldNames WHERE fieldName LIKE ‘prefix%;’

The second one would allow either suffixes or strings contained in another string:

SELECT fieldNames WHERE fieldName LIKE ‘%contained%;’

or

SELECT fieldNames WHERE fieldName LIKE ‘%suffix;’

As for the SPARSE mode, I just found here some details. This mode is mainly designed for cases when very few occurrences match the query. Based on this, I approached the definition of Query 3 as per the following cell. I create two indices working on the discipline and on the material type and another custom index for the email field. Then I build the WHERE clause on the three indices.

# set the indices  
session.execute('CREATE INDEX IF NOT EXISTS query3Index1 ON smalldb(discipline);')  
session.execute('CREATE INDEX IF NOT EXISTS query3Index2 ON smalldb(materialtype);')  
session.execute('CREATE CUSTOM INDEX IF NOT EXISTS SASIquery3Index3 ON smalldb(email) USING \'org.apache.cassandra.index.sasi.SASIIndex\' WITH OPTIONS = {\'mode\': \'CONTAINS\', \'analyzer\_class\': \'org.apache.cassandra.index.sasi.analyzer.NonTokenizingAnalyzer\', \'case\_sensitive\': \'false\'};')  
  
# define the CQL query  
test = 'SELECT materialid FROM smalldb WHERE discipline = \'maths\' AND materialtype = \'lecture slides\' AND email LIKE \'%gmail.com\' ALLOW FILTERING;'  
  
# run the query  
smallqueryTest\_query3 = session.execute(test)

---------------------------------------------------------------------------  
  
ReadFailure Traceback (most recent call last)  
  
Cell In [12], line 10  
 7 test = 'SELECT materialid FROM smalldb WHERE discipline = \'maths\' AND materialtype = \'lecture slides\' AND email LIKE \'%gmail.com\' ALLOW FILTERING;'  
 9 # run the query  
---> 10 smallqueryTest\_query3 = session.execute(test)  
  
  
File /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/cassandra/cluster.py:2637, in cassandra.cluster.Session.execute()  
  
  
File /Library/Frameworks/Python.framework/Versions/3.10/lib/python3.10/site-packages/cassandra/cluster.py:4920, in cassandra.cluster.ResponseFuture.result()  
  
  
ReadFailure: Error from server: code=1300 [Replica(s) failed to execute read] message="Operation failed - received 0 responses and 1 failures: UNKNOWN from /172.25.0.2:7000" info={'consistency': 'LOCAL\_ONE', 'required\_responses': 1, 'received\_responses': 0, 'failures': 1, 'error\_code\_map': {'172.25.0.2': '0x0000'}}

A ReadFailure error is thrown and I believe it is associated to the use of a custom index together with two regular indices. In fact, by executing two separate queries with the regular indices or the custom one, results are obtained.

test1 = 'SELECT materialid FROM smalldb WHERE discipline = \'maths\' AND materialtype = \'lecture slides\' ALLOW FILTERING;'  
queryTest1 = session.execute(test1)  
for i in range(0, 10):  
 print(queryTest1[i])

Row(materialid='21516')  
Row(materialid='4163')  
Row(materialid='21576')  
Row(materialid='21308')  
Row(materialid='4328')  
Row(materialid='21329')  
Row(materialid='21370')  
Row(materialid='21490')  
Row(materialid='21367')  
Row(materialid='21727')  
  
  
/var/folders/yb/85lnwyqj61q7l7rp510bt7340000gn/T/ipykernel\_1503/435765169.py:4: DeprecationWarning: ResultSet indexing support will be removed in 4.0. Consider using ResultSet.one() to get a single row.  
 print(queryTest1[i])

test2 = 'SELECT materialid FROM smalldb WHERE email LIKE \'%gmail.com\' ALLOW FILTERING;'  
queryTest2 = session.execute(test2)  
for i in range(0, 10):  
 print(queryTest2[i])

/var/folders/yb/85lnwyqj61q7l7rp510bt7340000gn/T/ipykernel\_1503/1889992488.py:4: DeprecationWarning: ResultSet indexing support will be removed in 4.0. Consider using ResultSet.one() to get a single row.  
 print(queryTest2[i])  
  
  
Row(materialid='17036')  
Row(materialid='9497')  
Row(materialid='11515')  
Row(materialid='9388')  
Row(materialid='19901')  
Row(materialid='23612')  
Row(materialid='8111')  
Row(materialid='13673')  
Row(materialid='11260')  
Row(materialid='18429')

I then resolved to run a more simplified query and manipulate the result via programming language to obtain the desired result. I get the rows where the discipline is maths and the learning material type is lecture slides, then I run a custom function on the query result. The function allows to obtain the domain of the email in the query result, this allows me to count only those matching the string ‘gmail.com’. In this case, all the query results are of interest, since we want to know how many learning materials have been accessed by students, irrespective of possible repetitions in accessed learning materials.

# define function taking an email string and returning a substring with the email domain  
def findDomain(email):  
 delimiter = '@'  
 emailList = email.split(delimiter)  
 return emailList[1]

# step 1  
small\_cassandra3 = 'SELECT email FROM smalldb WHERE discipline = \'maths\' AND materialtype = \'lecture slides\' ALLOW FILTERING;'  
  
before = time.time()  
small\_query3 = session.execute(small\_cassandra3)  
counter = 0  
for row in small\_query3:  
 if findDomain(row.email) == 'gmail.com':  
 counter += 1  
 else:  
 counter += 0  
after = time.time()  
  
print(counter)

837

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query3'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 small\_query3 = session.execute(small\_cassandra3)  
 counter = 0  
 for row in small\_query3:  
 if findDomain(row.email) == 'gmail.com':  
 counter += 1  
 else:  
 counter += 0  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query3'].append(round(msec\_duration, 5))

**Query 4**

Query 4 presents analogous problems to those of Query 3 since multiple occurrences of Korean countries are present in the table (South Korea, North Korea, Républica de Corea, etc.). In this case, to achieve the desired task, instead of trying to using a custom index, I preferred to exploit the IN set operator, by using it on the complete list of occurrences of Korean countries. In this way, only students from a country present in the limited set of Korean countries can be selected (together with those enrolled to a course of the discipline ‘psychology’). Considering that birthdate is simply another string value and given that string manipulation or pattern searches would require using a custom index together with other selecting approaches, which would raise the already experienced problems, choosing students born before year 2000 seems a hard task, which, given my configuration, would probably be better achieved via programming language. So I define a function to extract the year from the dateofbirth field and check if the year precedes year 2000. I use the function on the elements of the Python set to which I have already added the query results, which seems a more efficient approach than using the function on the query result and later adding surviving results in a Python set.

# define function taking a birthdate string in the format yyyy-mm-dd and returning a substring with the year  
def findYear(dateofbirth):  
 delimiter = '-'  
 dateList = dateofbirth.split(delimiter)  
 return int(dateList[0])

# step 1  
resSet4 = set()  
session.execute('CREATE INDEX IF NOT EXISTS query4Index1 ON smalldb(discipline);')  
small\_cassandra4 = 'SELECT firstname, lastname, country, dateofbirth FROM smalldb WHERE discipline = \'psychology\' AND country IN (\'Korea\', \'República de Corea\', \'South Korea\', \'North Korea\', \'República Popular Democrática de Corea\', \'Sydkorea\', \'Noord-Korea\') ALLOW FILTERING;'  
  
before = time.time()  
small\_query4 = session.execute(small\_cassandra4)  
for row in small\_query4:  
 if findYear(row.dateofbirth) < 2000:  
 resSet4.add(row)  
after = time.time()  
  
for element in resSet4:  
 print(element.firstname, element.lastname, element.country)

Raghav Sura North Korea  
Cathrine Lie South Korea  
Lynda Reynolds Korea

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query4'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 small\_query4 = session.execute(small\_cassandra4)  
 for row in small\_query4:  
 resSet4.add(row)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query4'].append(round(msec\_duration, 5))

smallDataset = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
for key in smallDict:  
 smallDataset[key].append(smallDict[key][0])  
 mean30 = mean(smallDict[key][1 : 31])  
 smallDataset[key].append(round(mean30, 5))  
smallDataset

{'query1': [4394.18602, 121.6824],  
 'query2': [1478.45817, 334.82787],  
 'query3': [317.37709, 270.71492],  
 'query4': [915.18497, 610.58322]}

The execution of the queries on the progressively larger datasets is omitted to avoid unnecessarily increasing the report length.

[…]

**MongoDB - Methodologies applied for loading the data and performing the queries**

**Preliminary operations: import csv files into MongoDB (mongoimport tool)**

There are basically two ways to import a csv file into a MongoDB instance. One is to use MongoDB Compass, an intuitive and comprehensive Graphical User Interface for MongoDB. The other is to use a MongoDB Database tool, i.e. mongoimport. The use of a GUI makes the import extremely user-friendly, although I prefer to implement a method that uses a standard command line API to import the data into the database. The mongoimport tool must be run from the system command line, not from the mongo shell, hence its execution must include host and authentication information to connect to the DBMS and interact with it.

#### **Syntax**

The mongoimport tool must be run from the system command line, not from within the MongoDB shell. Its syntax requires that we provide information for connecting to the desired MongoDB server (although by default the host name and port are the standard ones: localhost:27017), together with authentication details, database name and file details. Connection and authentication details can be provided through a connection string or via options. I opt for the latter method because it is more explicit and readable, in my opinion.

mongoimport [options] [connection-string] [file]

#### **Options**

- host name

The –host option allows us to indicate the hostname and port of the MongoDB instance we want to connect to. As previously specified, these need not be declared if the default ones (localhost:27017) are ok: mongoimport –host=localhost:27017

- authentication

If we specify a -password option (together with –username), we must also set the –authenticationDatabase option. These can all be specified explicitly in the command line, although it is recommended that the password is stored into a configuration file or inputed at the prompt. For the latter method, it is sufficient to let an empty string (’‘) follow the –password option. Username and password need not be enclosed in quotes. The –authenticationDatabase option must be provided when using the password option: it specifies the [authentication database](https://www.mongodb.com/docs/manual/core/security-users/#std-label-user-authentication-database) where the specified username has been created (admin, in my case).

mongoimport –host=localhost:27017 –username=root –password=’’

–authenticationDatabase=admin

- database

The –db option is used to declare the name of the database where we want to import the csv file. We can also use a –collection option to indicate the name of the collection within the previously declared database. If the –collection option is not used, mongoimport creates a collection within the declared database with the filename (without extension) as collection name.

mongoimport –host=localhost:27017 –username=root –password=’’

–authenticationDatabase=admin –db=dbB\_MONGODB\_test

- file

The file that must be imported must be specified together with a couple more options: -type needs to be specified if the file is not a JSON (default type), –headerline indicates that the first row of the csv contains the header, –file is for indicating the path and filename.

mongoimport –host=localhost:27017 –username=root –password=’’

–authenticationDatabase=admin –db=dbB\_MONGODB\_test

–type=csv –headerline –file=path/filename.csv

When the file is imported, mongoimport translates each row into a document, assigning to each value a key equal to the value of the corresponding column of the first row of the csv file. Since a csv file is flat, the resulting schema of the documents in the collection will be homogeneous.

#### **Notes when running MongoDB from a Docker container**

In loading the csv file from the local machine, we must take into account the usual feature of Docker virtual environments. They come with a file system of their own, so a DBMS run from within a container has access to this file system, not to the local machine file system. Hence, the csv file must be imported into the container via the usual docker cp command.

docker cp path/mycsvfile.csv container:/path

As previously pointed out, I just copy the file into the container’s root, without specifying a subfolder.Another important occurrence of running MongoDB within a Docker container regards the fact that we must execute the mongoimport tool from the system command line, not from the Mongo shell (mongosh). This means that we must run the mongoimport command as a Docker execution, without previously accessing the container:

docker exec container mongoimport –host=localhost:27017

–username=root –password=’’ –authenticationDatabase=admin

–db=dbB\_MONGODB\_test –type=csv –headerline

–file=‘filename.csv’

If importing is successful, before returning to the system folder from which the docker exec has been run, we are returned a message notifying the number of documents successfully imported (this should be equal to the number of rows of the csv file) and the number of those failed to import. By running mongosh from the container’s bash shell, or checking MongoDB Compass we can query the newly created collection.

**Performing the queries and storing the queries execution time**

#### **Python - MongoDB interaction**

Prior to performing the queries, we import the required modules (the PyMongo driver and the time and csv modules) and establish a connection to the MongoDB instance running in Docker. It is also convenient to store the database and the collections of interest into Python variables.

# import modules  
import pymongo # MongoDB driver  
import time # time-related functions to register query execution times  
import csv # read and write csv files  
  
# create client object  
client = pymongo.MongoClient(host = 'localhost', port = 27017, username = 'root', password = 'root')  
  
# assign database to Python variable (db)  
db = client.get\_database('dbB\_MONGODB\_test')  
  
# assign collections to Python variables  
smallColl = db.smallDB  
mediumColl = db.mediumDB  
largeColl = db.largeDB  
humongousColl = db.humongousDB

#### **Measuring and displaying the query execution time**

To display the query execution time we can use the Python time module and its time function. The function returns the system time at a floating point precision, so the query execution time can be measured as a large number of fractions of a second. It is sufficient to assign the time before the query execution to a variable and the time after the query execution to another variable. The difference between the two variables will measure the query execution. Obviously, the time for the Python API to connect to the MongoDB server and the time to return to the Python API after the query execution will be summed up to the query execution time at the DBMS level.

import time  
start = time.time()  
cursor1 = smallColl.find({'courseID' : 192}, {'\_id' : 0, 'studentID' : 1, 'firstName' : 1, 'lastName' : 1})  
end = time.time()  
print((end - start) \* 1000)

0.17881393432617188

#### **Query the collections**

I create a dictionary of lists for each of the four collections. In these dictionaries the keys are the query names and the values are the 31 query execution times: in fact I attach the value of the query execution time of the most recent query to the list. Since query execution times are required in milliseconds, prior to attaching them, I multiply them by 1000 and round them to the fifth decimal precision. The above summarized actions (for each of the four queries on each of the four collections) are performed by following a standard succession of steps. Each step is encapsulated within a notebook cell (so each query is performed 31 times by using three notebook cells), as follows:

* step 1: define the query, perform it for the first time, contextually create timestamps prior and after query execution, print query result;
* step 2: compute execution time of the first query execution and store it within the corresponding dictionary list;
* step 3: [thirty times] perform query execution while creating prior and following timestamps, compute execution time and store it within the corresponding dictionary list. With Mongo I don’t need to reset the cursor to allow repeating the query.

For each dataset, after having performed the four queries, I will finally compute the mean of the query executions from step 3. Together with the first query execution, this mean value will be stored into a new dictionary, specific to a dataset. Originally, I would use these four new dictionaries to save the query execution times into a csv file for constructing histograms. I later resolved to save all the 31 recorded query execution times and pass them all to Microsoft© Excel to process them.

smallDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
mediumDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
largeDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
humongousDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}

# mean function  
def mean(aList):  
 n = len(aList)  
 sum = 0  
 for value in aList:  
 sum += value  
 return sum / n

#### **Collection with 250k documents**

I start with the smallest collection.

**Query 1**

# step 1  
small\_mongo1 = [{'$match' : {'courseID' : 192}}, {'$group' : {'\_id' : {'name' : '$firstName', 'surname' : '$lastName'}}}]  
  
before = time.time()  
small\_query1 = smallColl.aggregate(small\_mongo1)  
after = time.time()  
  
for result in small\_query1:  
 print(result['\_id']['name'], result['\_id']['surname'])

Arthur Laroche  
Émile Nicolas  
Patrícia Leite  
Vigilija Gaižauskas  
Ingeborg Amundsen  
Sarah Lara  
Casandra Arenas  
Narciso Ferrán  
Cathrine Lie  
Custodia Hidalgo  
Ana Narušis  
Nath Nicolas  
Ledün Soylu

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query1'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 smallColl.aggregate(small\_mongo1)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query1'].append(round(msec\_duration, 5))

**Query 2**

# step 1  
small\_mongo2 = [{'$match' : {'discipline' : 'statistics', 'courseYear' : 2022}}, {'$group' : {'\_id' : {'ID' : '$courseID', 'course' : '$courseName'}}}, {'$project' : {'\_id.course' : 1}}]  
  
before = time.time()  
small\_query2 = smallColl.aggregate(small\_mongo2)  
after = time.time()  
  
for result in small\_query2:  
 print(result['\_id']['course'])

Econometrics: Methods and Applications  
Exploratory Data Analysis  
Introduction to Probability and Data with R  
Python and Statistics for Financial Analysis  
Basic Statistics  
Introduction to Statistics  
Foundations: Data, Data, Everywhere  
Understanding Clinical Research: Behind the Statistics  
Bayesian Statistics: From Concept to Data Analysis

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query2'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 smallColl.aggregate(small\_mongo2)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query2'].append(round(msec\_duration, 5))

**Query 3**

# step 1  
small\_mongo3 = [{'$match' : {'discipline': 'maths', 'materialType' : 'lecture slides', 'email': {'$regex': '@gmail.com'}}}, {'$group' : {'\_id': '\_id', 'IDcount' : {'$count': {}}}}]  
  
before = time.time()  
small\_query3 = smallColl.aggregate(small\_mongo3)  
after = time.time()  
  
for result in small\_query3:  
 print(result['IDcount'])

838

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query3'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 smallColl.aggregate(small\_mongo3)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query3'].append(round(msec\_duration, 5))

**Query 4**

# step 1  
small\_mongo4 = [{'$match' : {'discipline': 'psychology', 'country' : {'$regex': 'orea'}, 'dateOfBirth': {'$regex': '^1'}}}, {'$project' : {'\_id': 0, 'firstName': 1, 'lastName': 1, 'country': 1}}, {'$group' : {'\_id': {'name': '$firstName', 'surname': '$lastName', 'country': '$country'}}}, {'$sort' : {'lastName' : 1}}]  
  
before = time.time()  
small\_query4 = smallColl.aggregate(small\_mongo4)  
after = time.time()  
  
for result in small\_query4:  
 print(result['\_id']['name'], result['\_id']['surname'], 'from', result['\_id']['country'])

Raghav Sura from North Korea  
Lynda Reynolds from Korea  
Cathrine Lie from South Korea

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query4'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 smallColl.aggregate(small\_mongo4)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query4'].append(round(msec\_duration, 5))

smallDataset = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
for key in smallDict:  
 smallDataset[key].append(smallDict[key][0])  
 mean30 = mean(smallDict[key][1 : 31])  
 smallDataset[key].append(round(mean30, 5))  
smallDataset

{'query1': [1235.01515, 135.87454],  
 'query2': [192.60073, 169.13517],  
 'query3': [178.5481, 165.27492],  
 'query4': [172.63603, 157.43516]}

The execution of the queries on the progressively larger datasets is omitted to avoid unnecessarily increasing the report length.

[…]

**Redis - Methodologies applied for loading the data and performing the queries**

**Preliminary operations: import csv files into Redis**

Given Redis’ nature of a key-value store rather than a classical DBMS, importing is a task better performed via a programming language. This requires to load the csv file and store the data into a suitable data structure, then use the programming language to connect to a Redis instance and store the data into a Redis data type. Hence, this section on Redis will have a slightly different format from the previous ones, starting with describing how to manage csv files from Python and then introducing the Redis’ Python driver and the mothods used to connect to a Redis instance from Python.

#### **Importing csv files into Python**

Importing a csv file into Python is best performed via the csv module, contained in the Python standard library. It contains methods for reading (csv.reader) or writing (csv.writer) csv files. By starting a connection to a file we can read it line by line and store the fields within each line into lists. The file lines can be stored into a dictionary of lists where each line number corresponds to the dictionary key and the associated list contains the fields included in the csv lines. The above described procedure is stored into a function that can be called for each of the four datasets.

# PURPOSE: stores a csv file into a Python dictionary (dictionary keys are row numbers, values are rows as lists)  
# ARGUMENTS: a path (string), a csv filename including extension (string), a dictionary name  
# RETURNS: a dictionary  
def importCSVfile(pathName, csvFileName):  
 import csv  
 key = 0  
 dictName = dict()  
 with open(pathName + csvFileName, newline = '') as csvFile:  
 reader = csv.reader(csvFile, delimiter = ',')  
 for line in reader:  
 dictName[key] = line  
 key += 1  
 return dictName

# call the importCSVfile function for the four differently sized datasets  
path = '/Users/mau/OneDrive - unime.it/Learning/CdL Informatica/Anno II - Database/Module B/project/tables/'  
  
# 250k rows dataset to dict  
smallDB = importCSVfile(path, 'dataset250k.csv')  
  
# 500k rows dataset to dict  
mediumDB = importCSVfile(path, 'dataset500k.csv')  
  
# 750k rows dataset to dict  
largeDB = importCSVfile(path, 'dataset750k.csv')  
  
# 1m rows dataset to dict  
humongousDB = importCSVfile(path, 'dataset1m.csv')

print('Lengths of the four dictionaries, from the smallest to the largest:\n')  
print('smallDB:', len(smallDB), 'mediumDB:', len(mediumDB), 'largeDB:', len(largeDB), 'humongousDB:', len(humongousDB))

Lengths of the four dictionaries, from the smallest to the largest:  
  
smallDB: 250001 mediumDB: 500001 largeDB: 750001 humongousDB: 1000001

#### **Python - Redis interaction**

Interaction between a Python API and a Redis key-value store requires the installation of a specific driver. The usual list of drivers for various programming languages is provided in the Clients web page of the Redis website: redis-py is the driver developed by Redis Inc. for a Python programming environment.After having installed the driver, it can be imported into a Python environment the usual way.

import redis

Establishing a connection to a Redis database

We can connect to a Redis instance by simply assigning a Redis() object to a Python variable. By default, the driver sets a connection to a local Redis instance on port 6379. Host name and port can also be specified as arguments. By default, Redis returns responses as bytes in Python. We can be returned responses decoded as strings by specifying the decode\_responses argument.

myRedis = redis.Redis(host = 'localhost', port = 6379, decode\_responses = True)

I found it more convenient to access different Redis instances to perform the querying tests. This allows me to consider each instance as dedicated to a unique hash type (prefix). So each Redis instance could be conceptually treated as if it were an independent collection of documents. I could also store 2.5 million keys in a unique Redis instance and each hash prefix would allow identifying to which ‘collection’ each document belonged. To allow for the chosen implementation, I create three more Redis instances: in the first one (myRedis) I will store the smallest dataset, in the second one (myRedis2) I will store the dataset with 500k records, in the third one (myRedis3) I will store the dataset with 750k records and in the fourth one (myRedis4) I will store the dataset with 1m records.

myRedis2 = redis.Redis(host = 'localhost', port = 6382, decode\_responses = True)  
myRedis3 = redis.Redis(host = 'localhost', port = 6383, decode\_responses = True)  
myRedis4 = redis.Redis(host = 'localhost', port = 6384, decode\_responses = True)

A Redis connection implements a CoreCommands class which contains functions that can replicate all the commands provided within the redis-cli API. Since Python is case sensitive, however, they must be typed in the correct letter case (they usually use lowercase letters). The list of all available methods is accessible via the usual dir(redisObject) function.

#### **Store data into Redis hashes [ ! LONG PROCESSES FOLLOW ! ]**

Hashes are a Redis data type that allows the association of keys and values. A hash object has a name and a list of key-value stores. In our case, the keys may represent the field (column) names contained in the first row of the csv file (header) while the values are the field values contained in the other csv rows. We can create one hash per row by creating hash names of the form small:rownumber, where the text before the colon represents the dataset and the text after the colon is the row number. Thus, each row becomes a hash where the hash keys are common across all hashes in the dataset. This is helpful because hash keys may work as a schema for implementing queries. This procedure is stored into a function that takes a dictionary as argument, so that it is sufficient to feed it the desired dataset (one of the four differently sized dataset stored into the four dictionaries above) to have data sent to Redis (the process is quite long even for the 250k rows dataset, anyway).

# PURPOSE: stores Python dictionary key-value pairs to Redis hashes with a prefix  
# ARGUMENTS: a Redis instance, a dictionary(dict), a string we want to use as hash prefix  
# (usually the hash string ends with a colon to separate prefix and row number)  
# RETURNS: nothing  
def sendToRedis(redisInstance, datasetDict, hashPrefix):  
 for i in range(1, len(datasetDict)):  
 for j in range(0, len(datasetDict[0])):  
 redisInstance.hset(hashPrefix + str(i), datasetDict[0][j], datasetDict[i][j])

If we desire to remove hashes prefixed with the dataset name, sent to Redis as above explained, we can also reverse the process by looping over the length of the dataset dictionary (the dictionary keys range from 0 to 250k or 500k, etc.), assigning the dataset name + colon + the row number to the hashes we want to remove and applying the delete method on them. This will remove them one by one (a long process as well as the one of loading them). Again, we store the procedure into a function.

# PURPOSE: removes Redis hashes sent with a prefix from a Python dictionary  
# ARGUMENTS: a Redis instance, a dictionary(dict), a string we want to use as hash prefix  
# (usually the hash string ends with a colon to separate prefix and row number)  
# RETURNS: nothing  
def removeFromRedis(redisInstance, datasetDict, hashPrefix):  
 for i in range(1, len(datasetDict)):  
 redisInstance.delete(hashPrefix + str(i))

# THESE LINES START LONG PROCESSES, SO I COMMENT THEM OUT TO AVOID UNCAUTIOUS USE  
'''  
# 250k keys dict to Redis hashes  
sendToRedis(myRedis, smallDB, 'smallDB:')  
  
# 500k keys dict to Redis hashes  
sendToRedis(myRedis2, mediumDB, 'mediumDB:')  
  
# 750k keys dict to Redis hashes  
sendToRedis(myRedis3, largeDB, 'largeDB:')  
  
# 1m keys dict to Redis hashes  
sendToRedis(myRedis4, humongousDB, 'humongousDB:')  
'''

# optional delete hashes process  
# removeFromRedis(myRedis, mediumDB, 'mediumDB:')

We can consider each hash as a single document in a collection of documents where keys are common across them.

**Performing the queries and storing the queries execution time**

#### **Python - Redis interaction**

Prior to performing the queries, we import the required modules (the redis-py driver and the time and csv modules) and establish connections to the four different Redis instances running in Docker. Also, we import the set of dependencies that are necessary to perform queries and aggregations.

# import modules  
import redis # Redis driver (redis-py)  
import time # time-related functions to register query execution times  
import csv # read and write csv files  
  
# start four different Redis instances  
myRedis = redis.Redis(host = 'localhost', port = 6379, decode\_responses = True)  
myRedis2 = redis.Redis(host = 'localhost', port = 6382, decode\_responses = True)  
myRedis3 = redis.Redis(host = 'localhost', port = 6383, decode\_responses = True)  
myRedis4 = redis.Redis(host = 'localhost', port = 6384, decode\_responses = True)  
  
# import query-related redis-py dependencies  
from redis.commands.search.field import TextField, NumericField, TagField  
from redis.commands.search.indexDefinition import IndexDefinition, IndexType  
from redis.commands.search.query import Query  
import redis.commands.search.aggregation as aggregations  
import redis.commands.search.reducers as reducers

#### **Query the Redis instances**

I create a dictionary of lists for each of the four Redis instances. In these dictionaries the keys are the query names and the values are the 31 query execution times: in fact I attach the value of the query execution time of the most recent query to the list. Since query execution times are required in milliseconds, prior to attaching them, I multiply them by 1000 and round them to the fifth decimal precision. As previously explained, the above summarized actions (for each of the four queries on each of the four instances) are performed by following a standard succession of steps. Each step is encapsulated within a notebook cell (so each query is performed 31 times by using four notebook cells), as follows:

* step 1: define the query: create a RediSearch object, define the schema (basically, the indices needed for the specific query) and the index characteristics, create the index;
* step 2: perform the query for the first time, contextually create timestamps prior and after query execution, print query result;
* step 3: compute execution time of the first query execution and store it within the corresponding dictionary list;
* step 4: [thirty times] perform query execution while creating prior and following timestamps, compute execution time and store it within the corresponding dictionary list.

For each Redis instance, after having performed the four queries, I will finally compute the mean of the query executions from step 4. Together with the first query execution, this mean value will be stored into a new dictionary, specific to a dataset. Originally, I would use these four new dictionaries to save the query execution times into a csv file for constructing histograms. I later resolved to save all the 31 recorded query execution times and pass them all to Microsoft© Excel to process them.

N.B.: It must be noticed that, if an index has been already created, it resides in the memory of the system. Hence, either it is directly re-used or it must be deleted and created anew. Since I want to keep the code for documentation purposes, I keep a commented line with the instruction to destroy an index in each step1 cell for use when needed.

smallDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
mediumDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
largeDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
humongousDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}

# mean function  
def mean(aList):  
 n = len(aList)  
 sum = 0  
 for value in aList:  
 sum += value  
 return sum / n

#### **Redis instance with 250k hashes**

I start with the Redis instance with the smallest number of hash keys.

**Query1**

#step 1  
small\_redis1 = myRedis.ft('small\_index1')  
#small\_redis1.dropindex()  
schema1 = (TextField('courseID'), TextField('firstName'), TextField('lastName'))  
index\_definition = IndexDefinition(prefix = ['smallDB:'], index\_type = IndexType.HASH)  
small\_redis1.create\_index(schema1, definition = index\_definition)

'OK'

#step 2  
aggRequest1 = aggregations.AggregateRequest('@courseID:192').group\_by({'@firstName', '@lastName'})  
before = time.time()  
small\_query1 = small\_redis1.aggregate(aggRequest1)  
after = time.time()  
  
for res in small\_query1.rows:  
 print(res[1], res[3])

Ferrán Narciso  
Lie Cathrine  
Nicolas Émile  
Amundsen Ingeborg  
Laroche Arthur  
Lara Sarah  
Leite Patrícia  
Gaižauskas Vigilija  
Hidalgo Custodia  
Arenas Casandra  
Soylu Ledün  
Nicolas Nath  
Narušis Ana

#step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query1'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 small\_redis1.aggregate(aggRequest1)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query1'].append(round(msec\_duration, 5))

**Query2**

# step 1  
small\_redis2 = myRedis.ft('small\_index2')  
#small\_redis2.dropindex()  
schema2 = (TextField('discipline'), TextField('courseYear'), TextField('courseName'))  
index\_definition = IndexDefinition(prefix = ['smallDB:'], index\_type = IndexType.HASH)  
small\_redis2.create\_index(schema2, definition = index\_definition)

'OK'

# step 2  
aggRequest2 = aggregations.AggregateRequest('@discipline:statistics @courseYear:2022').group\_by('@courseName')  
before = time.time()  
small\_query2 = small\_redis2.aggregate(aggRequest2)  
after = time.time()  
  
for res in small\_query2.rows:  
 print(res[1])

Introduction to Probability and Data with R  
Basic Statistics  
Bayesian Statistics: From Concept to Data Analysis  
Python and Statistics for Financial Analysis  
Understanding Clinical Research: Behind the Statistics  
Econometrics: Methods and Applications  
Exploratory Data Analysis  
Foundations: Data, Data, Everywhere  
Introduction to Statistics

# step 3  
msec\_duration = (after - before) \* 1000  
smallDict['query2'].append(round(msec\_duration, 5))

# step 4  
for i in range(0, 30):  
 before = time.time()  
 small\_redis2.aggregate(aggRequest2)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query2'].append(round(msec\_duration, 5))

**Query3**

# step 1  
small\_redis3 = myRedis.ft('small\_index3')  
#small\_redis3.dropindex()  
schema3 = (TextField('materialType'), TagField('discipline'), TextField('email'), TextField('firstName'))  
index\_definition = IndexDefinition(prefix = ['smallDB:'], index\_type = IndexType.HASH)  
small\_redis3.create\_index(schema3, definition = index\_definition)

'OK'

# step 2  
aggRequest3 = aggregations.AggregateRequest('@discipline:{maths} @materialType:\'lecture slides\' @email:\*gmail.com').group\_by('@discipline', reducers.count().alias('count'))  
before = time.time()  
small\_query3 = small\_redis3.aggregate(aggRequest3)  
after = time.time()  
  
print(small\_query3.rows[0][3])

838

# step 3  
msec\_duration = (after - before) \* 1000  
smallDict['query3'].append(round(msec\_duration, 5))

# step 4  
for i in range(0, 30):  
 before = time.time()  
 small\_query3 = small\_redis3.aggregate(aggRequest3)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query3'].append(round(msec\_duration, 5))

**Query4**

# step 1  
small\_redis4 = myRedis.ft('small\_index4')  
#small\_redis4.dropindex()  
schema4 = (TagField('discipline'), TagField('courseYear'), TextField('country'), TextField('dateOfBirth'), TextField('firstName'), TextField('lastName', sortable = True))  
index\_definition = IndexDefinition(prefix = ['smallDB:'], index\_type = IndexType.HASH)  
small\_redis4.create\_index(schema4, definition = index\_definition)

'OK'

#step 2  
aggRequest4 = aggregations.AggregateRequest('@discipline:{psychology} AND @courseYear:{2023} AND @country:\*orea AND -@dateOfBirth:200\*').group\_by({'@firstName', '@lastName', '@country', '@dateOfBirth'}).sort\_by('@lastName')  
before = time.time()  
small\_query4 = small\_redis4.aggregate(aggRequest4)  
after = time.time()  
  
for res in small\_query4.rows:  
 print(res[1], res[5], res[7], res[3])

lie South Korea Cathrine 1986-7-12  
reynolds Korea Lynda 1989-7-21  
sura North Korea Raghav 1973-11-27

# step 3  
msec\_duration = (after - before) \* 1000  
smallDict['query4'].append(round(msec\_duration, 5))

# step 4  
for i in range(0, 30):  
 before = time.time()  
 small\_redis4.aggregate(aggRequest4)  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query4'].append(round(msec\_duration, 5))

smallDataset = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
for key in smallDict:  
 smallDataset[key].append(smallDict[key][0])  
 mean30 = mean(smallDict[key][1 : 31])  
 smallDataset[key].append(round(mean30, 5))  
smallDataset

{'query1': [43.05792, 6.59162],  
 'query2': [12.70509, 19.69381],  
 'query3': [17.30895, 17.22561],  
 'query4': [14.01019, 7.57892]}

The execution of the queries on the progressively larger datasets is omitted to avoid unnecessarily increasing the report length.

[…]

**Neo4j - Methodologies applied for loading the data and performing the queries**

**Preliminary operations: import csv files into Neo4j (LOAD CSV)**

A csv file can be imported into Neo4j via the LOAD CSV command, described in the Cypher manual (Cypher is Neo4j’s declarative query language), although other methods are better suitable for large files.

#### **Syntax**

The file to be imported via the LOAD CSV command is preceded by a file:/// prefix. This prefix points to the import directory in the neo4j home folder (for security reasons this directory is the default root for files that are imported via LOAD CSV). The csv file must be copied/moved to the import folder prior to importing.

LOAD CSV FROM ‘<file:///mycsvfile.csv>’ … ;

The LOAD CSV command loads data line by line: each line is treated as an array of strings (it is to be considered that the imported data are always read as strings, hence if we want to have them as a different data type within our Neo4j instance, we should use conversion tools available within the LOAD CSV command) and each field is an element of the array. Indexing the line array gives us access to a specific field value. For example, if each line of our csv file contains the attributes of various entities, entities can be abstracted from their attributes and created as labeled nodes, and their attributes can be considered as node properties. If the schema is known and rigid, it is possible to know the attributes of each entity within the data: indexing the line array, thus, allows to distribute the properties to the nodes we create from abstracting the entities.

LOAD CSV FROM ‘<file:///mycsvfile.csv>’ AS line

CREATE(:node1 {property1: line[0], property2: line[1], …,

property[i]: line[j]}),

(:node2 {property1: line[m], property2: line[n], …,

property[p]: line[q]}), … ;

#### **Options**

Options permit to take into account the specific structure of different csv files. They help in adapting to csv files with a header row or to specify a custom field delimiter (the default is a comma).

- WITH HEADERS

The presence of a header row in the csv file can improve the process of node and property creation. If this is the case, we can include the WITH HEADERS option in the LOAD CSV statement and field value access can be performed via field name rather than by indexing:

LOAD CSV WITH HEADERS FROM ‘<file:///mycsvfile.csv>’ AS line

CREATE(:node1 {property1: line.fieldName, property2:

line.fieldName, …, property[i]: line.fieldName}),

(:node2 {property1: line.fieldName, property2:

line.fieldName, …, property[p]: line.fieldName}), … ;

- FIELDTERMINATOR

Fields in the line array are separated by a comma by default, but this setting can be overridden by the FIELDTERMINATOR option:

LOAD CSV WITH HEADERS FROM ‘<file:///mycsvfile.csv>’ AS line

FIELDTERMINATOR ‘,’

CREATE(:node1 {property1: line.fieldName, property2:

line.fieldName, …, property[i]: line.fieldName}),

(:node2 {property1: line.fieldName, property2:

line.fieldName, …, property[p]: line.fieldName}), … ;

#### **Actual LOAD CSV statement implementation**

According to the above presentation, a further consideration needs to be done on the actual implementaion that needs to be performed.It is, in fact, advised to separate node and relationship creation into separate processing: this way the load is only doing one piece of the import at a time and can move through large amounts of data quickly and efficiently, reducing heavy processing. The actual statement passed to Neo4j for importing the dataset with 250k rows is then the following:

* Create student nodes:

LOAD CSV WITH HEADERS FROM ‘<file:///dataset250k.csv>’ AS line MERGE(:student {studentID: line.studentID, firstName:

line.firstName, lastName: line.lastName, dob:

line.dateOfBirth, genre: line.genre, country:

line.country, town: line.town, email: line.email});

* Create course nodes:

LOAD CSV WITH HEADERS FROM ‘<file:///dataset250k.csv>’ AS line MERGE(:course {courseID: line.courseID, discipline: line.discipline, courseName: line.courseName, courseYear: line.courseYear, syllabus: line.syllabus});

* Create learning material nodes:

LOAD CSV WITH HEADERS FROM ‘<file:///dataset250k.csv>’ AS line MERGE(:material {materialID: line.materialID, unit: line.unit,

mType: line.materialType, mName: line.name, dimension:

line.dimension, accessDate: line.accessDate});

* Create relationship between student nodes and course nodes:

LOAD CSV WITH HEADERS FROM ‘<file:///dataset250k.csv>’ AS line

MATCH(s:student {studentID: line.studentID, firstName:

line.firstName, lastName: line.lastName, dob:

line.dateOfBirth, genre: line.genre, country:

line.country, town: line.town, email: line.email})

MATCH(c:course {courseID: line.courseID, discipline:

line.discipline, courseName: line.courseName,

courseYear: line.courseYear, syllabus:

line.syllabus}) MERGE (s) -[:is\_enrolled]-> (c);

* Create relationship between student nodes and learning material nodes:

LOAD CSV WITH HEADERS FROM ‘<file:///dataset250k.csv>’ AS line MATCH(s:student {studentID: line.studentID, firstName:

line.firstName, lastName: line.lastName, dob:

line.dateOfBirth, genre: line.genre, country:

line.country, town: line.town, email: line.email})

MATCH(m:material {materialID: line.materialID, unit: line.unit,

mType: line.materialType, mName: line.name, dimension:

line.dimension, accessDate: line.accessDate}) MERGE (s) –

[:studies] -> (m);

* Create relationship between course nodes and learning material nodes:

LOAD CSV WITH HEADERS FROM ‘<file:///dataset250k.csv>’ AS line MATCH(c:course {courseID: line.courseID, discipline:

line.discipline, courseName: line.courseName, courseYear:

line.courseYear, syllabus: line.syllabus}) MATCH(m:material {materialID: line.materialID, unit: line.unit,

mType: line.materialType, mName: line.name, dimension:

line.dimension, accessDate: line.accessDate}) MERGE (c) –

[:uses] -> (m);

#### **A note on dealing with memory overhead**

When using the LOAD CSV method, if the csv file size is large and a large amount of nodes or relationships are to be created, it is possible to run into memory allocation problems. There are different ways to address this kind of problems, I have attempted two of them, one which exploits the apoc.periodic.iterate procedure from the APOC Neo4j plugin (some references here), the other one which uses the CALL … IN TRANSACTIONS clause.

1. APOC plugin

As for the APOC plugin, it can be installed by conveniently setting the required environment variables in the Docker compose file (see Neo4j Docker compose file used in the project). The procedure can be called by using the CALL command and it requires three parameters:

* a first Cypher statement to retrieve the data (in my case, for example I need a first statement with two MATCH commands to retrieve the required nodes and a RETURN command with the nodes variable names);
* a second Cypher statement to tell the system what to do with the data (create the relationship between the retrieved nodes, in my case, by using the MERGE command);
* a final parameter (in the form of a dictionary of key-value pairs) to set various options such as the batch size, which will determine how many steps will be used to perform the procedure.

One example is the following:

CALL apoc.periodic.iterate(

“LOAD CSV WITH HEADERS FROM ‘<file:///dataset500k.csv>’ AS line

MATCH(s:student {studentID: line.studentID, firstName:

line.firstName, lastName: line.lastName, dob:

line.dateOfBirth, genre: line.genre, country: line.country, town: line.town, email: line.email})

MATCH(m:material {materialID: line.materialID, unit:

line.unit, mType: line.materialType, mName: line.name, dimension: line.dimension}) RETURN s, m”,

“MERGE (s) -[:studies {accessDate: line.accessDate}] -> (m)”,

{batchSize: 5000, parallel: false});

1. CALL … IN TRANSACTIONS

As for the CALL … IN TRANSACTIONS subquery, it follows the LOAD CSV clause and tells the system to commit after a given number of rows are processed. The previous example would be written as follows:

LOAD CSV WITH HEADERS FROM ‘<file:///dataset500k.csv>’ AS line CALL { WITH line MATCH(s:student {studentID: line.studentID, firstName: line.firstName, lastName: line.lastName, dob: line.dateOfBirth, genre: line.genre, country: line.country, town: line.town, email: line.email}) MATCH(m:material {materialID: line.materialID, unit: line.unit, mType: line.materialType, mName: line.name, dimension: line.dimension}) RETURN s, m”, MERGE (s) -[:studies {accessDate: line.accessDate}] -> (m) } IN TRANSACTIONS OF 500 ROWS;

The methodology I chose for the project is the second one.

#### **Notes when running Neo4j from a Docker container**

In loading the csv file from the local machine, we must take into account the fact that Docker virtual environments come with a file system of their own, so a DBMS run from within a container has access to this file system, not to the file system of the local machine. Hence, the csv file must be imported into the container via the usual docker cp command.

docker cp path/mycsvfile.csv container:/path

For security reasons, there is a default directory from which it is allowed to import external files (the import directory). So it is advisable to set this directory as docker cp destination, instead of the container root (to change this setting, the config file must be edited, although it is not recommended). The command syntax would then be:

docker cp path/mycsvfile.csv container:/var/lib/neo4j/import

**Performing the queries and storing the queries execution time**

#### **Python - Neo4j interaction**

Prior to performing the queries, I import the required modules (the Neo4j Python driver and the time and csv modules) and set four driver objects to which I pass the parameters to establish a connection to each of the four Neo4j instances running in Docker.

# import modules  
from neo4j import GraphDatabase # Neo4j driver  
import time # time-related functions to register query execution times  
import csv # read and write csv files  
  
# create driver object  
driver = GraphDatabase.driver(uri = 'neo4j://127.0.0.1:7687', auth = ('neo4j', 'myPassword'))  
driver2 = GraphDatabase.driver(uri = 'neo4j://127.0.0.1:7692', auth = ('neo4j', 'myPassword'))  
driver3 = GraphDatabase.driver(uri = 'neo4j://127.0.0.1:7693', auth = ('neo4j', 'myPassword'))  
driver4 = GraphDatabase.driver(uri = 'neo4j://127.0.0.1:7694', auth = ('neo4j', 'myPassword'))

#### **Query the Neo4j instances**

I create a dictionary of lists for each of the four Neo4j instances. In these dictionaries the keys are the query names and the values are the 31 query execution times: in fact I attach the value of the query execution time of the most recent query to the list. Since query execution times are required in milliseconds, prior to attaching them, I multiply them by 1000 and round them to the fifth decimal precision. As previously explained, the above summarized actions (for each of the four queries on each of the four instances) are performed by following a standard succession of steps. Each step is encapsulated within a notebook cell (so each query is performed 31 times by using three notebook cells), as follows:

* step 1: define the query as a string and pass it together with the database parameter to the execute\_query() method of the driver object, while contextually registering the time before and after the query execution. Finally print query results;
* step 2: compute execution time of the first query execution and store it within the corresponding dictionary list;
* step 3: [thirty times] perform query execution while creating prior and following timestamps, compute execution time and store it within the corresponding dictionary list.

For each Neo4j instance, after having performed the four queries, I will finally compute the mean of the query executions from step 3. Together with the first query execution, this mean value will be stored into a new dictionary, specific to a dataset. Originally, I would use these four new dictionaries to save the query execution times into a csv file for constructing histograms. I later resolved to save all the 31 recorded query execution times and pass them all to Microsoft© Excel to process them.

N.B.: Contrary to what has been introduced when presenting the methodologies used for accessing and querying Neo4j, since I don’t need to use the information contained in the summary object to get the query execution time, I am only interested in the query result records for showing them. I can use dot notation on the query object to access the query records (a query object has a records attribute). I could use the index 0 on the query object since the records are the first element in a list. By iterating on the records object and applying the data() method during iteration I can turn the records elements into key-value pairs (a dictionary, actually).

smallDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
mediumDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
largeDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
humongousDict = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}

# mean function  
def mean(aList):  
 n = len(aList)  
 sum = 0  
 for value in aList:  
 sum += value  
 return sum / n

#### **Neo4j instance with 250k hashes**

I start with the smallest Neo4j instance.

**Query 1**

# step 1  
small\_neo4j1 = 'MATCH (s:student) -[:is\_enrolled]-> (c:course) WHERE c.courseID = \'192\' RETURN s.firstName, s.lastName'  
  
before = time.time()  
small\_query1 = driver.execute\_query(small\_neo4j1, database = 'neo4j')  
after = time.time()  
  
records = small\_query1.records  
for record in records:  
 print(record.data()['s.firstName'], record.data()['s.lastName'],)

Custodia Hidalgo  
Ledün Soylu  
Nath Nicolas  
Ana Narušis  
Émile Nicolas  
Sarah Lara  
Patrícia Leite  
Cathrine Lie  
Arthur Laroche  
Casandra Arenas  
Narciso Ferrán  
Vigilija Gaižauskas  
Ingeborg Amundsen

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query1'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 driver.execute\_query(small\_neo4j1, database = 'neo4j')  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query1'].append(round(msec\_duration, 5))

**Query 2**

# step 1  
small\_neo4j2 = 'MATCH(c:course) WHERE c.discipline = \'statistics\' AND c.courseYear = \'2022\' RETURN c.courseName'  
  
before = time.time()  
small\_query2 = driver.execute\_query(small\_neo4j2, database = 'neo4j')  
after = time.time()  
  
records = small\_query2.records  
for record in records:  
 print(record.data()['c.courseName'])

Econometrics: Methods and Applications  
Exploratory Data Analysis  
Understanding Clinical Research: Behind the Statistics  
Introduction to Probability and Data with R  
Bayesian Statistics: From Concept to Data Analysis  
Introduction to Statistics  
Python and Statistics for Financial Analysis  
Basic Statistics  
Foundations: Data, Data, Everywhere

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query2'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 driver.execute\_query(small\_neo4j2, database = 'neo4j')  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query2'].append(round(msec\_duration, 5))

**Query 3**

# step 1  
small\_neo4j3 = 'MATCH (c:course {discipline: \'maths\'}) <-[:is\_enrolled]- (s:student) -[:studies]-> (m:material {mType: \'lecture slides\'}) <-[:uses]- (c:course) WHERE right(s.email, 9) = \'gmail.com\' RETURN COUNT(m);'  
  
before = time.time()  
small\_query3 = driver.execute\_query(small\_neo4j3, database = 'neo4j')  
after = time.time()  
  
records = small\_query3.records  
for record in records:  
 print(record.data()['COUNT(m)'])

838

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query3'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 driver.execute\_query(small\_neo4j3, database = 'neo4j')  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query3'].append(round(msec\_duration, 5))

**Query 4**

# step 1  
small\_neo4j4 = 'MATCH (s:student) -[:is\_enrolled]-> (c:course {discipline: \'psychology\'}) WHERE right(s.country, 4) = \'orea\' AND left(s.dob, 1) <> \'2\' RETURN DISTINCT s.firstName, s.lastName, s.country ORDER BY s.lastName;'  
  
before = time.time()  
small\_query4 = driver.execute\_query(small\_neo4j4, database = 'neo4j')  
after = time.time()  
  
records = small\_query4.records  
for record in records:  
 print(record.data()['s.firstName'], record.data()['s.lastName'], record.data()['s.country'])

Cathrine Lie South Korea  
Lynda Reynolds Korea  
Raghav Sura North Korea

# step 2  
msec\_duration = (after - before) \* 1000  
smallDict['query4'].append(round(msec\_duration, 5))

# step 3  
for i in range(0, 30):  
 before = time.time()  
 driver.execute\_query(small\_neo4j4, database = 'neo4j')  
 after = time.time()  
 msec\_duration = (after - before) \* 1000  
 smallDict['query4'].append(round(msec\_duration, 5))

smallDataset = {'query1' : list(), 'query2' : list(), 'query3' : list(), 'query4' : list()}  
for key in smallDict:  
 smallDataset[key].append(smallDict[key][0])  
 mean30 = mean(smallDict[key][1 : 31])  
 smallDataset[key].append(round(mean30, 5))  
smallDataset

{'query1': [5489.99786, 24.541],  
 'query2': [418.41102, 17.52477],  
 'query3': [1242.91325, 39.00357],  
 'query4': [449.72205, 14.5945]}

The execution of the queries on the progressively larger datasets is omitted to avoid unnecessarily increasing the report length.

[…]

**Experiment results**

In this final section I introduce by means of a graphical representation the test results concerning the execution time of the four queries on each of the differently sized dataset. I start with the smallest dataset (250 thousand *records*) and then move to those larger in size (up to the one with 1 million *records*). I present the results for each of the five DBMS used for the tests, each one characterized by a different colour, to mark the differences between one another. They are presented in the same order across each of the four plots that follow (MySQL, Cassandra, MongoDB, Redis and Neo4j). Here I only present the average result of the 30 executions following the first one, together with the related 95% confidence intervals. First results are presented for each DBMS individually in the attached spreadsheet.

#### **Dataset with 250k records**

The different results in execution time are already evident in applying the four queries to the smallest dataset. While the performances of MySQL, Cassandra and MongoDB are comparable, those of Redis and Neo4j are significantly lower than the other three. Also, while MySQL execution times increase with the query complexity, the NoSQL DBMSs do not show the same behavior. In particular, both Redis and Neo4j are faster in performing the third query rather than the fourth. Finally, Redis’ and Neo4j’s confidence intervals are significantly lower than those of the other DBMSs: with 95% confidence, the query result will be very close to the reported average.

#### **Dataset with 500k records**

The same patterns described for the previous dataset are more or less present for the one with 500 thousand record. MySQL’s query execution times increase with the query complexity. MongoDB behaves increasingly efficiently, something that is due, perhaps, to the goodness of its methodology of presenting the execution of the more complex queries as a series of stages. It also show very narrow confidence intervals, so the reported mean should represent the true execution time with a good level of confidence. Redis and Neo4j are the two DBMSs with the best performance even in this case, with very little confidence intervals.

#### **Dataset with 750k records**

The execution times seem to increase as expected with the increasing number of records in the dataset: this, at least, is the pattern of MySQL and MongoDB (both systems showing narrow confidence intervals), while Redis and Neo4j seem faster in running the queries for the dataset with 750 thousand records rather than for the one with 500 thousand records.

#### **Dataset with 1m records**

Again, MySQL and MongoDB show higher execution times with the increasing records of the dataset, but their average performance can be trusted with good confidence. Generally, Redis and Neo4j have shown a better performance in running query 4 rather than query 3, and this is also the case for the largest dataset. Redis’ average execution time is associated with a confidence interval larger than usual, and in fact the average execution time for query 3 seems too high.

**Conclusions**

To conclude, a few words must be dedicated to illustrating the performance of Cassandra, which I didn’t mention before. Cassandra behaves far better than MySQL and significantly better than MongoDB in running query 1, while it performs worse than both in running query 2. The results for query 3 are not consistent across the datasets: Cassandra behaves better on the datasets with 500 thousand and 750 thousand records, while it performs worse on the smallest and on the largest datasets. Finally, its performance is very bad in running query 4 on all of the datasets considered. Usually Cassandra is considered a very fast dataset, due to its design which encourages constructing the tables having in mind the expected queries. In this project, I could not exploit in full Cassandra’s capacities in that I tried to maintain a unique data representation on all of the datasets. With respect to this, although conceptually very different from DBMSs that base data representation on table design, Redis and Neo4j have demonstrated an incredible flexibility in adapting their data models to the data structure used in the project. Turning back to Cassandra, its negative performance is mostly due to the strategy to let part of the steps necessary to obtain the desired results be handled by the programming language, adding a layer of complexity which negatively impacts on the overall efficiency of the process.